

**A n t h o n y   J o n e s**

**J i m   O h i u n d**

**N e t   w o r k**

**P r o g r a m m i n g**

**f o r**

**Microsoft®**

**W i n d o w s**

***M i c r o s o f t   P r e s s***

Э. Джонс, Д. Оланд

# Программирование в сетях Master<sup>®</sup>

# Windows

и Linux:

# МАСТЕР-КЛАСС

«.РУССКАЯ РЕДАКЦИЯ

о Москва • Санкт-Петербург • ХарШсов • Минск  
2002

УДК 004.43  
Ж 32.973.26-018

иконе Э., Оланд Дж.

Программирование в сетях Microsoft Windows. Мастер-класс. / Пер. с англ. —  
СПб.: Питер, М: Издательско-торговый дом «Русская Редакция», 2002. — 608 стр.: ил.

ISBN 5-318-00725-2  
ISBN 5-7502-0148-1

Книга знакомит читателя с многообразием сетевых функций ОС семейства Windows. Обсуждается разработка сетевых приложений на платформе Win32 с использованием интерфейсов программирования NetBIOS и Winsock, а также распространенных протоколов. На конкретных примерах рассмотрены клиент-серверная модель; установка соединения и передача данных; регистрация и разрешение имен, в том числе применительно к Windows 2000 и Active Directory; широковещание в сети; ATM; QoS и удаленный доступ. В приложениях содержится справочник команд NetBIOS (с указанием входных и выходных параметров), сведения о новых функциях IP Helper, а также справочник кодов ошибок Winsock.

Адресована как профессиональным программистам, так и новичкам, для которых станет удобным справочником и исчерпывающим пособием по использованию сетевых функций Windows

Состоит из 15 глав, трех приложений и предметного указателя; прилагаемый компакт-диск содержит примеры программ.

УДК 004.43  
ББК 32.973.26-018

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США

Intel — охраняемый товарный знак компании Intel Corporation Active Directory, ActiveX, Authenticode, BackOffice, BizTalk, JSnpt, Microsoft, Microsoft Press, MSDN, MSN, NetMeeting, Outlook, Visual Basic, Win32, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах Все другие товарные знаки являются собственностью соответствующих фирм

Все названия компаний, организаций и продуктов, а также имена лиц и события, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам, событиям и лицам, если особо не указано иное

Г

© Оригинальное издание на английском языке, Microsoft Corporation, 2000  
© Перевод на русский язык, Microsoft Corporation, 2001

ISBN 0-7356-0560-2 (англ.)  
ISBN 5-318-00725-2  
' 5-7502-0148-1

# Оглавление

Введение	XII
<b>Ч А С Т Ь I. УСТАРЕВШИЕ СЕТЕВЫЕ API</b>	<b>1</b>
<b>ГЛАВА 1. Интерфейс NetBIOS</b>	<b>2</b>
Интерфейс Microsoft NetBIOS	3
Номера LANA	4
Имена NetBIOS	5
Особенности NetBIOS	8
Основы программирования NetBIOS	9
Синхронный и асинхронный вызов	11
Типичные процедуры NetBIOS	11
Сервер сеансов: модель асинхронного обратного вызова	19
Сервер сеансов: модель асинхронных событий	24
Клиент сеанса NetBIOS	30
Дейтаграммные операции	34
Дополнительные команды NetBIOS	48
Проверка состояния адаптера (команда <i>NCBASTAT</i> )	49
Команда поиска имени ( <i>NCBFINDNAME</i> )	51
Сопоставление протоколов номерам LANA	51
Рекомендации по выбору платформ	52
Платформа Windows CE	52
Платформа Windows 9x	52
Для любых платформ	53
Резюме	53
<b>ГЛАВА 2. Перенаправитель</b>	<b>54</b>
Универсальные правила именования	55
Поставщик нескольких UNC	55
Компоненты сетевого доступа	56
Перенаправитель	57
Протокол SMB	57
Безопасность	59
Дескрипторы безопасности	59
Маркеры доступа	61
Сетевая безопасность	61
Реквизиты сеанса	61
Пример	62
Резюме	63
<b>ГЛАВА 3. Почтовые ящики</b>	<b>64</b>
Подробности внедрения почтовых ящиков	64
Имена почтовых ящиков	65
Л* Размеры сообщений	65
ОС  Компиляция приложения	67
Коды ошибок	67

Общие сведения об архитектуре клиент-сервер	67
Сервер почтовых ящиков	67
Клиент почтовых ящиков	70
1.дополнительные API-функции почтовых ящиков	72
Платформа и производительность	73
Правила именования < 8 3 >	73
Неспособность отменить блокирующие запросы ввода-вывода	74
Утечки памяти	76
Резюме	77
<b>"ЛАВА 4. Именованные каналы</b>	<b>78</b>
Детали реализации именованных каналов	79
Правила именования каналов	79
Режимы побайтовый и сообщений	79
Компиляция приложений	79
Коды ошибок	80
Простой сервер и клиент	80
Детали реализации сервера	80
Усовершенствованный сервер каналов	87
Детали реализации клиента	95
Другие API-вызовы	98
Платформа и производительность	101
Резюме	101
<b>Ч А С Т Ь II. ИНТЕРФЕЙС ПРИКЛАДНОГО</b>	
<b>ПРОГРАММИРОВАНИЯ WINSOCK</b>	<b>103</b>
<b>"ЛАВА 5. Сетевые протоколы</b>	<b>104</b>
Характеристики протоколов	104
Протокол, ориентированный на передачу сообщений	104
Обмен данными, с соединением и без него	106
Надежность и порядок доставки сообщений	106
Скоректное завершение работы	107
Широковещание данных	108
Многоадресное вещание	108
Качество обслуживания	108
Фрагментарные сообщения	109
Маршрутизация	109
Другие характеристики	109
Поддерживаемые протоколы	110
Сетевые протоколы, поддерживаемые Win32	110
Сетевые протоколы в Windows CE	112
Информация о протоколе	112
Зокеты Windows	116
Простые сокеты	118
Информация о платформах	118
Winsock и модель OSI	119
Выбор соответствующего протокола	119
Резюме	120

ГЛАВА 6. Семейства адресов и разрешение имен	121
Протокол IP	121
Протокол TCP	121
Протокол UDP	122
Адресация	122
Порядок байт	124
Создание сокета	125
Разрешение имен	125
Инфракрасные сокеты	128
Адресация	128
Разрешение имен	128
Нумерация IrDA-устройств	129
Опрос IAS	131
Создание сокета	133
Параметры сокета	133
Протоколы IPX/SPX	133
Адресация	133
Создание сокета	134
Разрешение имен	137
Протоколы NetBIOS	137
Адресация	137
Создание сокета	139
Протокол AppleTalk	140
Адресация	140
Создание сокета	148
Протокол АТМ	148
Адресация	149
Создание сокета	153
Привязка сокета к SAP	154
Разрешение имен	155
Дополнительные функции Winsock 2	155
Резюме	156
ГЛАВА 7. Основы Winsock	157
Инициализация Winsock	157
Проверка и обработка ошибок	159
Протоколы с установлением соединения	160
Серверные API-функции	160
API-функции клиента	164
Передача данных	168
Потоковые протоколы	173
Завершение сеанса	175
Пример	176
Протоколы, не требующие соединения	185
Приемник	185
Отправитель	187
Протоколы, ориентированные на передачу сообщений	188
Освобождение ресурсов сокета	189
Пример	189

1,ополнительные функции API	197
Функция <i>getpeername</i>	197
Функция <i>getsockname</i>	198
Функция <i>WSADuphcateSocket</i>	198
Функция <i>TransmitFile</i>	199
ля платформы Windows CE	200
ругие семейства адресов	201
Протокол AppleTalk	201
Инфракрасные сокет	202
Интерфейс с NetBIOS	202
Протокол IPX/SPX	203
Протокол ATM	204
'езюме	204
"ЛАВА 8. Ввод-вывод в Winsock	205
'ежимы работы сокет	206
Блокирующий режим	206
Неблокирующий режим	208
Додел	209
Модель <i>select</i>	209
Модель <i>WSAAsyncSelect</i>	213
Модель <i>WSAEventSelect</i>	217
Модель перекрытого ввода-вывода	223
Модель портов завершения	234
'равнение моделей ввода-вывода	243
Клиент	243
Сервер	243
<sup>5</sup> езюме	243
"ЛАВА 9. Параметры сокета и команды управления СВОИМ-ВЫВОДОМ	245
Тараметры сокета	245
Уровень <i>SOLJOCKET</i>	246
Уровень параметров <i>SOLAPPLETALK</i>	255
Уровень параметров <i>SOL IRLMP</i>	258
Уровень параметров <i>IPPROTOJP</i>	262
Уровень параметров <i>IPPROTOTCP</i>	267
Уровень параметров <i>NSPROTO IPX</i>	268
Функции <i>Ioctlsocket</i> и <i>WSAlotl</i>	272
Стандартные ioctl-команды	273
Другие ioctl-команды	274
Ioctl-команды Secure Socket Layer	282
Ioctl-команды для ATM	283
Резюме	285
"ЛАВА 10. Регистрация и разрешение имен	286
введение	286
Модели пространства имен	287
Перечень пространств имен	287
Регистрация службы	289

Определение класса службы	289
Регистрация службы	293
Запрос к службе	299
Создание запроса	301
Запрос к DNS	304
Резюме	307
<b>ГЛАВА 11. Многоадресная рассылка</b>	<b>308</b>
Семантика многоадресной рассылки	308
Свойства многоадресной рассылки	311
Многоадресная рассылка в сетях IP	311
Протокол IGMP	312
Листовые узлы IP	313
Реализация IP-рассылки	314
Многоадресная рассылка в сетях ATM	314
Листовые узлы ATM	315
Корневые узлы ATM	315
Многоадресная рассылка с использованием Winsock	316
Рассылка средствами Winsock 1	316
Рассылка средствами Winsock 2	323
Общие параметры Winsock	340
Ограничение многоадресной рассылки при удаленном доступе	342
Резюме	342
<b>ГЛАВА 12. Качество обслуживания</b>	<b>343</b>
Введение	343
Протокол RSVP	344
Сетевые компоненты	344
Компоненты приложения	346
Компоненты политики безопасности	347
QoS и Winsock	348
Структуры QoS	349
Функции, вызывающие QoS	352
Завершение QoS	356
Объекты, относящиеся к поставщику	356
Программирование QoS	365
RSVP и типы сокетов	366
Уведомления QoS	368
Шаблоны QoS	371
Примеры	373
Одноадресный TCP	373
Одноадресный UDP	394
Многоадресный UDP	395
ATM и QoS	396
Резюме	397
<b>ГЛАВА 13. Простые сокеты</b>	<b>398</b>
Создание простого сокета	398
Протокол ICMP	399
Пример Ping	401



Программа Traceroute.....	411
Протокол IGMP.....	412
Использование <i>IPJiDRINCL</i> .....	414
Резюме.....	424
<b>"ЛАВА 14. Интерфейс Winsock 2 SPI</b>	<b>425</b>
Основа SPI.....	426
Соглашения SPI об именах.....	426
Соответствие функций Winsock 2 API и SPI.....	426
Поставщики транспортной службы.....	427
Функция <i>WSPStartup</i> .....	428
Описатели сокетов.....	433
Поддержка модели ввода-вывода Winsock.....	435
Модель <i>select</i> .....	437
Расширенные функции.....	446
Установка поставщиков транспортной службы.....	447
Поставщики службы пространства имен.....	453
Установка поставщика пространства имен.....	453
Реализация пространства имен.....	455
Пример.....	461
Отладочные функции отслеживания Winsock 2 SPI.....	466
Резюме.....	467
<b>"ЛАВА 15. Элемент управления Winsock</b>	<b>468</b>
Свойства.....	468
Методы.....	470
События.....	471
Пример (UDP-приложение).....	472
Пересылка UDP-сообщений.....	476
Прием UDP-сообщений.....	477
Получение информации от элемента Winsock.....	478
Запуск UDP-приложения.....	478
Состояние UDP-сокетов.....	479
Пример (TCP-приложение).....	480
СР-сервер.....	487
TCP-клиент.....	489
Получение информации о состоянии элемента управления Winsock.....	490
Запуск TCP-приложения.....	490
Состояние TCP-сокетов.....	491
Ограничения.....	491
Типичные ошибки.....	493
Ошибка Local address in use.....	493
Ошибка Invalid Operation at Current State.....	493
Элемент управления Windows CE Winsock.....	494
Пример.....	494
Проблема с элементом управления VBCE Winsock.....	499
Резюме.....	499

<b>Часть III. Служба удаленного доступа (RAS)</b>	<b>500</b>
<b>Глава 16. Клиент службы RAS</b>	<b>501</b>
Компиляция и компоновка	502
Структуры данных и вопросы совместимости платформ	503
Обновление DUN 13 и Windows 95	503
Функция <i>RasDial</i>	503
Синхронный режим	506
Асинхронный режим	507
Уведомление о состоянии	512
Завершение соединения	513
Телефонный справочник	514
Добавление записей в телефонный справочник	522
Переименование записи телефонного справочника	525
Удаление записей из телефонного справочника	525
Перечисление записей телефонного справочника	526
Управление реквизитами пользователя	527
Многоканальные подзаписи телефонного справочника	529
Управление соединением	530
Резюме	534
<b>Приложение А. Перечень команд NetBIOS</b>	<b>535</b>
<b>Приложение В. Вспомогательные функции IP</b>	<b>549</b>
Возможности утилиты <i>Ipsconfig</i>	549
Освобождение и обновление IP-адресов	553
Изменение IP-адреса	554
Возможности утилиты <i>Netstat</i>	554
Получение таблицы TCP-соединений	555
Получение таблицы прослушиваемых портов UDP	556
Получение статистики о протоколе IP	557
Возможности утилиты <i>Route</i>	561
Получение таблицы маршрутов	562
Добавление маршрута	564
Удаление маршрута	565
Утилита <i>ARP</i>	565
Добавление записи ARP	567
Удаление записи ARP	567
<b>Приложение С. Коды ошибок Winsock</b>	<b>568</b>
От авторов	575
Предметный указатель	576

# Введение

Перед вами книга, посвященная сетевым функциям Windows 9x, NT 4, 2000 и CE. Она предназначена в первую очередь опытным программистам и специалистам по сетям. Впрочем, для начинающих она может послужить полезным справочным пособием и даже вводным курсом по сетевым функциям.

## О чем эта книга

В книге три части, посвященные работе в сети с применением NetBIOS и перенаправителя Windows, Winsock и RAS соответственно.

В главе 1 рассматривается NetBIOS. По нашему опыту работы в команде поддержки разработчиков Microsoft мы знаем, что многие компании все еще используют эту технологию. Между тем, до сих пор нет адекватного руководства по написанию приложений NetBIOS для платформ Win32. В главе 1 так же приводятся методы написания надежных и переносимых приложений (с учетом, что многие разработчики используют NetBIOS именно для связи с устаревшими системами).

Главы 2-4 посвящены перенаправителю Windows, почтовым ящикам и именованным каналам. Как вы знаете, почтовые ящики и именованные каналы основаны на перенаправителе. Мы решили посвятить перенаправителю целую главу, чтобы предоставить читателю базовую информацию о том, как три эти технологии соотносятся друг с другом. Почтовые ящики — это ненадежный односторонний ориентированный на сообщения интерфейс прикладного программирования, не зависящий от доступных в системе протоколов. Именованные каналы обладают более широкими возможностями, обеспечивая надежную двустороннюю дейтаграммную или поточную передачу данных. Эти каналы используют средства безопасности Windows NT за счет перенаправителя, на что не способен ни один другой сетевой API-интерфейс.

Вторая часть книги посвящена API-интерфейсу Winsock. Глава 5 — это введение в Winsock, где рассказывается о наиболее распространенных протоколах Winsock. Все приложения Winsock должны создавать сокет для осуществления связи. В этой главе мы приводим основную информацию о возможностях каждого протокола, а в главе 6 — подробное описание, как создать сокет и разрешить имя для каждого типа протокола.

В главе 7 — самое интересное. Здесь мы представляем базовую модель программирования клиент-сервер и описываем большинство функций Winsock, которые относятся к установлению и приему связи, передаче данных и т.п. Далее, в главе 8 рассказывается о методах ввода-вывода в Winsock. Так как глава 7 задумывалась как введение в указанную тему, в ней обсуждаются только простейшие методы ввода-вывода. В главе 8, напротив, эти методы описаны под-

робно Если вы новичок в работе с Winsock, то главы 5-7 помогут вам овладеть основами использования этого API-интерфейса

Остальные главы этой части книги посвящены особым аспектам и возможностям Winsock. Параметры сокетов и команды управления вводом-выводом рассмотрены в главе 9. Именно здесь вы найдете описание большинства команд, влияющих на работу сокета или даже протокола. Надеюсь, эта глава будет полезна, как в учебных, так и в справочных целях.

В главе 10 рассмотрена регистрация и разрешение имен служб в адреса базового протокола в Winsock 2. Это независимый от протокола метод. Распространение Windows 2000 и Active Directory придает данной главе особую значимость.

Глава 11 посвящена связи «точка — много точек»>, включая многоадресное IP-вещание и ATM. В главе 12 описана захватывающая технология — Quality of Service (QoS), позволяющая гарантировать выделение пропускной способности сети для приложений. В главе 13 рассказывается о простых IP-сокетах, мы рассматриваем, как приложения Winsock могут использовать их для работы с протоколами ICMP и IGMP, а также другие аспекты программирования при помощи простых сокетов.

В главе 14 описан интерфейс поставщика службы для Winsock — средства, при помощи которого программист может задать уровень между Winsock и поставщиками служб более низких уровней (например, TCP/IP) для управления работой сокета и протокола, регистрацией и разрешением имен. Этот сложный инструмент позволяет расширить функциональность Winsock.

И наконец, в главе 15 обсуждается элемент управления Microsoft Visual Basic для Winsock. Мы решили включить эту главу в книгу, так как убедились, что многие разработчики до сих пор полагаются на Visual Basic и этот элемент. Функциональность элемента Winsock ограничена и не позволяет использовать дополнительные новые свойства Winsock, но он незаменим для тех, кому требуется простая и легкая в использовании сетевая связь в Visual Basic.

Часть III посвящена клиентскому серверу удаленного доступа (Remote Access Server, RAS). Мы решили включить в книгу главу о RAS из-за популярности Интернета и широкого распространения коммутируемого доступа к нему. Возможность коммутируемого доступа в сетевом приложении очень полезна, так как упрощает работу пользователя с программой. То есть конечному пользователю не придется думать, как установить соединение, чтобы работать с сетевым приложением.

В конце книги — три приложения. Приложение А — справочник по командам NetBIOS, который, по нашему мнению, для программистов бесценен. В нем перечислены параметры ввода и вывода для каждой команды. В приложении В описаны новые вспомогательные функции IP, выдающие полезную информацию о сетевой конфигурации текущего компьютера. Приложение С — справочник по кодам ошибок Winsock с подробным описанием отдельных ошибок и возможных причин их возникновения.

Мы надеемся, что наша работа станет для вас ценным учебным и справочным пособием. Думаем, что это наиболее полная книга о сетевом программировании для Windows.

## Как пользоваться прилагаемым компакт-дискom

В тексте книги мы часто приводим примеры программ, иллюстрирующие работу с обсуждаемыми сетевыми API-интерфейсами. Эти примеры записаны на прилагаемый компакт-диск. Для их установки вставьте компакт-диск в дисковод, и программа Autorun запустит программу установки. Программу установки можно также инициировать вручную, запустив файл PressCD.exe из корневого каталога компакт-диска. Вы вправе установить образцы кода на компьютер или работать с ними прямо с компакт-диска (из папки Examples\Chapters\Chapter XX).

**ПРИМЕЧАНИЕ** Для работы с компакт-дискom необходима 32-битная ОС Windows.

Наряду с примерами программ в состав компакт-диска включена последняя версия Microsoft Platform SDK. Мы сделали это, потому что многие из наших примеров рассчитаны на современные заголовочные файлы и библиотеки, которые появились только после Windows 2000 Beta 3.

## Поддержка

Авторы приложили все усилия, чтобы обеспечить точность содержания книги и прилагаемого к ней компакт-диска. Издательство Microsoft Press публикует постоянно обновляемый список исправлений и дополнений к своим книгам по адресу <http://mspress.microsoft.com/support/>.

Многие определения функций и таблицы в книге адаптированы или переизданы с разрешения и при активном участии группы документирования Microsoft Platform SDK. Часть материала основана на предварительно разработанной документации и может претерпеть изменения. Информацию, обновления и исправления ошибок по последней версии SDK см. на Web-узле MSDN по адресу: <http://msdn.microsoft.com/developer/sdk/platform.asp>.

Если все же у вас возникнут вопросы или вы захотите поделиться своими предложениями или комментариями, обращайтесь в издательство Microsoft Press по одному из этих адресов:

*[mspinput@microsoft.com](mailto:mspinput@microsoft.com)*

Microsoft Press

Attn: *Network Programming for Microsoft Windows* Editor

One Microsoft Way

Redmond, WA 98052-6399

# УСТАРЕВШИЕ СЕТЕВЫЕ API

о

Первая часть издания посвящена сетевому интерфейсу NetBIOS, перенаправителю и типам использующих его сетевых соединений. Хотя в книге в основном обсуждается программирование средствами Winsock, мы включили в нее часть I, так как устаревшие сетевые API имеют некоторые преимущества перед Winsock.

В главе 1 рассматривается интерфейс NetBIOS, который, как и Winsock, является независимым от протокола сетевым API. NetBIOS обеспечивает асинхронные вызовы, а также совместимость со старыми операционными системами типа OS/2, DOS и др. В главе 2 обсуждается перенаправитель, с которым связаны две следующие темы: почтовые ящики (глава 3) и именованные каналы (глава 4). Перенаправитель обеспечивает независимый от транспорта ввод-вывод файлов. Почтовые ящики — это простой интерфейс, который помимо прочего поддерживает широковещание и однонаправленное взаимодействие между компьютерами под управлением Windows. Наконец, именованные каналы дают возможность использовать двусторонний канал связи, который поддерживает функции безопасности Windows.

## Интерфейс NetBIOS

Network Basic Input/Output System (NetBIOS) — стандартный *интерфейс прикладного программирования* (application programming interface, API), разработанный Sytek Corporation для IBM в 1983 г. NetBIOS определяет программный интерфейс для сетевой связи, но не обуславливает физический способ передачи данных по сети. В 1985 г. IBM предприняла попытку сформировать цельный протокол — создала NetBIOS Extended User Interface (NetBEUI), интегрированный с интерфейсом NetBIOS. Программный интерфейс NetBIOS вскоре приобрел такую популярность, что поставщики ПО начали реализовать его для других протоколов, таких как TCP/IP и IPX/SPX. В настоящее время NetBIOS используют платформы и приложения во всем мире, включая многие компоненты Windows NT, Windows 2000, Windows 95 и Windows 98.

**ПРИМЕЧАНИЕ** Windows CE не дает возможности использовать NetBIOS API, хотя поддерживает транспортный протокол TCP/IP, имена NetBIOS и механизм их разрешения.

В Win32 интерфейс NetBIOS обеспечивает обратную совместимость со старыми приложениями. В этой главе обсуждаются главные принципы программирования с помощью NetBIOS, начиная с имен NetBIOS и номеров LANA. Мы рассмотрим основные услуги, предлагаемые NetBIOS, включая те, что обеспечивают связь с установлением соединения или без такового (дейтаграммные). В каждом разделе будет приведен простой клиент-серверный пример. Завершит главу обзор типичных ошибок. В приложении А вы найдете список команд NetBIOS с обязательными параметрами и кратким описанием.

### Сетевая модель OSI

Модель Open Systems Interconnect (OSI) обеспечивает высокоуровневое представление сетевых систем. Ее семь уровней полностью описывают фундаментальные сетевые концепции: от приложения до способа физической передачи данных. Вот эти уровни:

**I прикладной** — предоставляет пользовательский интерфейс для передачи данных между программами;

- **представительский** — форматирует данные;

**III сеансовый** — управляет связью между двумя узлами;

- III транспортный** — обеспечивает передачу данных (надежную или ненадежную);
- И сетевой** — поддерживает механизм адресации между узлами и маршрутизацию пакетов данных;
  - **канальный** — управляет взаимодействием между узлами на физическом уровне; отвечает за группировку данных, передаваемых по физическому носителю;
- II физический** — физический носитель, ответственный за передачу данных в виде электрических сигналов.

В этой модели NetBIOS относится к сеансовому и транспортному уровням.

## Интерфейс Microsoft NetBIOS

Как уже упоминалось, существуют реализации NetBIOS API для разных сетевых протоколов, что делает интерфейс независимым от протокола. Иначе говоря, если вы разработали приложение согласно спецификации NetBIOS, оно может использовать протоколы TCP/IP, NetBEUI или даже IPX/SPX. Эта полезная особенность позволяет корректно написанному приложению NetBIOS выполняться почти на любом компьютере, независимо от физической сети. Однако есть несколько нюансов. Чтобы два приложения NetBIOS могли связаться друг с другом по сети, они должны выполняться на рабочих станциях, имеющих по крайней мере один общий транспортный протокол. Например, если на компьютере Джона установлен только TCP/IP, а на компьютере Мэри — только NetBEUI, приложения NetBIOS на компьютере Джо не смогут связаться с приложениями на компьютере Мэри.

Кроме того, только определенные протоколы реализуют интерфейс NetBIOS. Например, Microsoft TCP/IP и NetBEUI делают это по умолчанию, а IPX/SPX — нет. Поэтому Microsoft предлагает реализующую этот интерфейс версию IPX/SPX, что следует учесть при проектировании сети. При установке протоколов обычно видно, поддерживает ли версия протокола IPX/SPX возможности NetBIOS. Например, вместе с Windows 2000 поставляется NWLink IPX/SPX/NetBIOS Compatible Transport Protocol — то, что этот протокол поддерживает NetBIOS, прямо вытекает из его названия. В Windows 95 и Windows 98 в окне свойств протокола IPX/SPX есть флажок, с помощью которого включается поддержка NetBIOS для IPX/SPX.

Важно, что NetBEUI — немаршрутизируемый протокол. Если между клиентом и сервером есть маршрутизатор, приложения на этих компьютерах не смогут связаться. Маршрутизатор будет отбрасывать пакеты по мере их приема. TCP/IP и IPX/SPX — маршрутизируемые протоколы и не имеют такого ограничения. Так что если вы твердо запланировали использовать NetBIOS, задействуйте в сети по крайней мере один из маршрутизируемых транспортных протоколов. Характеристики протоколов и рекомендации по их выбору мы приводим в главе 6.



## Номера LANA

Какое отношение имеют транспортные протоколы к NetBIOS с точки зрения программирования? Ключ к пониманию NetBIOS — *номера сетевых адаптеров* (LAN Adapter, LANA). В первоначальных реализациях NetBIOS каждому физическому сетевому адаптеру присваивалось уникальное значение — номер LANA. В Win32 это стало проблематичным, так как рабочая станция может иметь и множество сетевых протоколов, и множество плат сетевого интерфейса.

Номер LANA соответствует уникальным сочетаниям сетевого адаптера с транспортным протоколом. Так, если рабочая станция имеет две сетевых платы и два поддерживающих NetBIOS транспорта (например, TCP/IP и NetBEUI), будет присвоено четыре номера LANA. Номера могут соответствовать сочетаниям адаптера с протоколом примерно следующим образом:

- 0 — «TCP/IP — сетевой адаптер 1»;
- 1 — «NetBEUI — сетевой адаптер 1»;
- 2 — «TCP/IP — сетевой адаптер 2»;
- 3 — «NetBEUI — сетевой адаптер 2».

Номера LANA лежат в диапазоне от 0 до 9, и операционная система назначает их без какого-либо определенного порядка. Кроме LANA 0, который имеет особый смысл — это номер «по умолчанию». Когда появился интерфейс NetBIOS, большинство операционных систем поддерживало единственный номер LANA и многие приложения были жестко запрограммированы на работу только с LANA 0. Для обратной совместимости вы можете вручную назначить LANA 0 конкретному протоколу.

В Windows 95 и Windows 98 можно открыть диалоговое окно свойств сетевого протокола с помощью значка Network в Control Panel. Выберите вкладку Configuration в диалоговом окне Network, затем из списка компонентов — сетевой протокол и щелкните кнопку Properties. Вкладка Advanced диалогового окна свойств для каждого протокола, поддерживающего NetBIOS, содержит флажок Set This Protocol To Be The Default Protocol. Пометка флажка перестраивает привязки протоколов так, чтобы протоколу по умолчанию был назначен LANA 0. Этот флажок можно пометить только для одного протокола. Поскольку Windows 95 и Windows 98 поддерживают Plug-and-Play, не требуется явно задавать приоритет протоколов.

Windows NT 4 допускает большую гибкость в установке NetBIOS. На вкладке Services диалогового окна Network выберите NetBIOS Interface из списка Network Services и щелкните кнопку Properties. В диалоговом окне NetBIOS Configuration вы можете явно назначить номера LANA всем сочетаниям сетевого интерфейса с транспортным протоколом. В этом диалоговом окне сетевой интерфейс определяется именем его драйвера. Впрочем, имена протоколов не всегда говорят сами за себя. На рис. 1-1 показано диалоговое окно NetBIOS Configuration. Оно свидетельствует: на компьютере установлено два сетевых адаптера и три транспортных протокола — TCP/IP (NetBT), NetBEUI (Nbf) и IPX/SPX (NwlnkNb). Щелкнув кнопку Edit, вы можете вручную назначить номера LANA для отдельных протоколов.

**NetBIOS Configuration**

Use this screen to change the lane number on the listed NetBIOS network adapters

```

i>erj NetworkBisute " " " "
000 Nbf->IEEPRO->IEEPRO2 - " ,
001 NetBT -> IEEPRO -> IEEPRO2
002 NwinkNb > Nwinklpx
003 Nbf->E100B -> E100B1
004 NetBT->E100B->E100B1

```

**Рис. 1-1. Диалоговое окно NetBIOS Configuration**

Windows 2000 также позволяет напрямую назначать номера LANA. В Control Panel щелкните значок Network And Dial-up Connections. Далее выберите в меню Advanced команду Advanced Settings и в открывшемся диалоговом окне настройте параметры на вкладке LANA numbers.

При разработке устойчивого приложения NetBIOS всегда пишется код, который может обрабатывать соединения на любом номере LANA. Предположим, Мэри пишет серверное приложение NetBIOS, которое слушает клиентов на LANA 2. На компьютере Мэри LANA 2 соответствует протоколу TCP/IP. Далее Джон решает написать клиентское приложение для связи с сервером Мэри, так что его приложение будет связываться через LANA 2 на его рабочей станции; однако, LANA 2 на компьютере Джона соответствует NetBEUI. Приложения не смогут связаться друг с другом, хотя им доступны протоколы TCP/IP и NetBEUI. Чтобы устранить это несоответствие, серверное приложение Мэри должно слушать клиентские соединения на каждом доступном номере LANA на рабочей станции Мэри. Аналогично, клиентское приложение Джона должно пытаться связаться на каждом номере LANA, доступном на его компьютере. Так Мэри и Джон смогут гарантировать успех связи их приложений. Конечно, наличие кода, который может обрабатывать соединения на любом номере LANA не означает, что этот код будет работать, если у двух компьютеров не найдется ни одного общего протокола.

## Имена NetBIOS

Теперь перейдем к именам NetBIOS. Процесс — или, если угодно, приложение — регистрирует имя на каждом номере LANA, с которым ему требуется связаться. Имя NetBIOS имеет длину 16 символов; 16-й символ зарезервирован для специальных целей. При добавлении имени в таблицу имен, вы должны очистить буфер имен. В среде Win32 каждый процесс имеет таблицу имен NetBIOS для каждого доступного номера LANA. Добавление имени для LANA 0 означает, что приложение доступно только клиентам, соединяющимся на этом LANA 0. Максимально к каждому номеру LANA могут быть Добавлены 254 имени, они пронумерованы от 1 до 254 (0 и 255 зарезерви-

рованы для системы) Впрочем, каждая ОС задает максимальный номер по умолчанию, меньший 254 Вы вправе его изменить при переопределении каждого номера LANА

Есть два типа имен NetBIOS уникальное и групповое Никакой другой процесс в сети не может зарегистрировать уже имеющееся уникальное имя будет выдана ошибка дублирования имени Как вам, наверное, известно, имена компьютеров в сетях Microsoft — имена NetBIOS Когда компьютер загружается, он регистрирует свое имя на локальном сервере Windows Internet Naming Server (WINS), который сообщает об ошибке, если другой компьютер уже использует то же имя Сервер WINS поддерживает список всех зарегистрированных имен NetBIOS

Вместе с именем могут храниться и сведения о протоколе Например, в сетях TCP/IP WINS запоминает IP-адрес компьютера, зарегистрировавшего имя NetBIOS Если сеть сконфигурирована без сервера WINS, компьютеры проверяют, нет ли в сети такого же имени, путем широковещательной рассылки сообщения Если никакой другой компьютер не оспаривает сообщение, сеть позволяет отправителю использовать заявленное имя

Групповые имена используются, чтобы отправлять или, наоборот, получать данные, предназначенные для множества получателей Имя группы не обязательно должно быть уникальным Групповые имена используются для многоадресной рассылки

16-й символ в именах NetBIOS определяет большинство сетевых служб Microsoft Имена служб и групп для WINS-совместимых компьютеров сервер WINS регистрирует напрямую, а для регистрации имен других компьютеров применяется широковещание по локальной подсети Чтобы получить информацию о зарегистрированных на локальном (или удаленном) компьютере именах NetBIOS, используйте утилиту Nbtstat В табл 1-1 приведены сведения о зарегистрированных именах NetBIOS, которые команда Nbtstat -n выдала для пользователя Davemac, вошедшего в компьютер, сконфигурированный как основной контроллер домена и работающий под управлением Windows NT Server вместе с Internet Information Server

**Табл. 1-1. Таблица имен NetBIOS**

Имя	16-й байт	Тип имени	Служба
DAVEMAC 1	<00>	Уникальное	Имя службы <i>рабочей</i> станции
DAVEMAC 1	<20>	Уникальное	Имя службы сервера
DAVEMACD	<00>	Групповое	Имя домена
DAVEMACD	<1C>	Групповое	Имя контроллера домена
DAVEMACD	<1B>	Уникальное	Имя координатора сети
DAVEMAC 1	<03>	Уникальное	Имя отправителя
Inet~Services	<1C>	Групповое	Групповое имя Internet Information Server
IS~DAVEMAC1	<00>	Уникальное	Уникальное имя Internet Information Server
DAVEMAC1+++++++		<BF>	Уникальное имя сетевого монитора

Утилита Nbtstat устанавливается вместе с протоколом TCP/IP. Она может также опрашивать таблицу имен на удаленных компьютерах, используя параметр -a с именем удаленного компьютера или параметр -A с его IP-адресом.

Перечислим стандартные значения 16-го байта, добавляемые в конец уникальных NetBIOS-имен компьютеров разными сетевыми службами Microsoft.

*III* <00> — имя службы рабочей станции (NetBIOS-имя компьютера),

- <03> — имя службы сообщений, используемое при получении и отправке сообщений, зарегистрировано сервером WINS для службы сообщений на клиенте WINS и обычно добавляется в конец имени компьютера и вошедшего в систему пользователя,
- <1B> — имя координатора сети (master browser) домена, определяет основной контроллер домена и указывает, каких клиентов и других обозревателей использовать для контакта с координатором сети домена,

*Я* <06> — серверная служба удаленного доступа (RAS),

- <1F> — служба сетевого динамического обмена данными (Network Dynamic Data Exchange, NetDDE),

*III* <20> — имя службы на сервере, используемое для предоставления точек подключения к общим файлам,

- <21> — клиент RAS,
- <BE> — агент сетевого монитора,
- <BF> — утилита Network Monitor (Сетевой монитор)

А теперь перечислим заданные по умолчанию символы 16-го байта, добавляемые в конец обычно используемых групповых имен NetBIOS.

- <1C> — групповое имя домена, содержащее список определенных адресов компьютеров, которые его зарегистрировали. Контроллер домена это имя регистрирует. WINS обрабатывает его как доменную группу. Каждый член группы должен индивидуально обновить свое имя или будет исключен. Доменная группа ограничена 25 именами. Если в результате репликации статическое 1C-имя конфликтует с динамическим 1C-именем на другом сервере WINS, для соответствующих участников добавляется комбинированная запись, которая помечается как статическая. Если запись статическая, члены группы не должны обновлять свои IP-адреса.

*III* <1D> — имя координатора сети (master browser), используемое клиентами для обращения к нему. В подсети может быть лишь один координатор. Серверы WINS возвращают положительный ответ на регистрацию имени домена, но не сохраняют это имя в своих базах данных. Если компьютер посылает запрос на имя домена серверу WINS, тот возвращает отрицательный ответ. Если компьютер, сделавший запрос, сконфигурирован как h- или m-узел, он затем выполняет широковещательную рассылку запроса имени, чтобы разрешить его. Тип узла определяет способ разрешения имени клиентом. Клиенты, сконфигурированные для разрешения в режиме b-узла, выполняют широковещательную рассылку пакетов для оповещения о себе и разрешения имен NetBIOS. При разрешении в ре-

жиме р-узла связь с сервером WINS осуществляется в стиле «точка — точка». При разрешении имен в режиме m-узла (смешанном) сначала задействуется режим b-узла, а затем при необходимости — режим р-узла. Последний метод разрешения — h-узел (гибридный). В этом режиме сначала применяется регистрация и разрешение в режиме р-узла, а в случае неудачи — в режиме b-узла. По умолчанию в Windows используется режим п-узла.

*щ* <1E> — обычное групповое имя. Обозреватели могут выполнять широко-вещательную рассылку для данного имени и слушать на нем, чтобы выбрать координатора сети. Эти широковещательные рассылки эффективны лишь в локальной подсети и не ретранслируются маршрутизаторами.

*Ш* <20> — имя Интернет-группы. Регистрируется серверами WINS, чтобы идентифицировать группы компьютеров в административных целях. Например, можно зарегистрировать групповое имя «printersg» для идентификации административной группы серверов печати.

*Ш* MSBROWSE — добавляется в конец имени домена вместо 16-го символа. Это имя широковещательно рассылается по локальной подсети, чтобы сообщить о домене другим ее координаторам.

Такое количество спецификаторов может казаться чрезмерным: скорее всего, вам не потребуется использовать их в именах NetBIOS, но все же примите их к сведению. Во избежание случайных коллизий между именами NetBIOS не используйте спецификаторы уникальных имен. Еще более осторожно относитесь к групповым именам: если ваше имя будет противоречить существующему групповому имени, ошибка выдана не будет, и вы начнете получать данные, предназначенные для кого-то другого.

## Особенности NetBIOS

NetBIOS предлагает и требующие логического соединения службы, и службы без установления соединения (дейтаграммные). Первые позволяют двум объектам устанавливать сеанс или виртуальный канал между ними. Сеанс — двусторонний коммуникационный поток, по которому объекты обмениваются сообщениями. Требующие сеанса службы гарантируют доставку любых данных между двумя конечными точками. Сервер обычно регистрирует себя под определенным известным именем, и клиенты ищут это имя, чтобы связаться с сервером. Применительно к NetBIOS, процесс сервера добавляет его имя в таблицу имен для каждого номера LAN, с которым требуется связь. Клиенты на других компьютерах преобразуют имя службы в имя компьютера и затем запрашивают соединение у серверного процесса. Как видите, для установления такой цепочки необходимо предварительно выполнить определенные действия, поэтому инициализация соединения связана с некоторыми издержками. При установлении сеанса гарантируется доставка пакетов и их порядок, хотя связь и основана на сообщениях. То есть если подключенный клиент направит команду чтения, сервер вернет только один пакет данных в потоке, даже если клиент обеспечил буфер для приема нескольких пакетов.

При использовании служб, не требующих логического соединения (дейтаграммных), сервер регистрируется под конкретным именем, и клиент просто собирает данные и посылает их в сеть, не устанавливая заранее никакого соединения. Клиент направляет данные на NetBIOS-имя серверного процесса. Дейтаграммные службы эффективнее, поскольку не тратят время и ресурсы на установление соединения.

С другой стороны, дейтаграммные службы не дают гарантий доставки и порядка сообщений. Например, клиент отправляет тысячи байтов данных на сервер, который два дня назад был остановлен из-за отказа. Отправитель никогда не получит никаких уведомлений об ошибке, если только не ожидает ответа от сервера (если в течение достаточно длительного периода времени нет отклика, то можно понять, что что-то не так).

## Основы программирования NetBIOS

Теперь обсудим API-интерфейс NetBIOS. Он элементарен, поскольку содержит только одну функцию:

```
UCHAR Netbios(PNCB pNCB);
```

Все объявления функций, константы и т. п. для NetBIOS определены в заголовочном файле. Единственная библиотека, необходимая для компоновки приложений NetBIOS — Netapi32.lib. Наиболее важная особенность этой функции — параметр *pNCB*, который является указателем на *блок сетевого управления* (network control block, NCB). Это — указатель на структуру *NCB*, содержащую всю информацию, требуемую функции *Netbios* для выполнения команды NetBIOS. Определяется эта структура так:

```
typedef struct _NCB

{
    '  UCHAR      ncb_command;
'-  UCHAR      ncb_retcode;
,  UCHAR      ncb_lsn;
JJ  UCHAR      ncb_num;
,  PCHAR      ncb.buffer;
    WORD       ncb_length;
    UCHAR      ncb_callname[NCBNAMSZ];
    UCHAR      ncb_name[NCBNAMSZ];
    UCHAR      ncb_rto;
4   UCHAR      ncb.sto;
1   void       (*ncb_post) (struct _NCB *);
    UCHAR      ncb_lana_num;
,   UCHAR      ncb_cmd_cplt;
    UCHAR      ncb_reserve[10];
    HANDLE     ncb_event;
\ * PNCB, NCB;
```

Не все члены структуры будут использоваться в каждом вызове NetBIOS; некоторые из полей данных являются выходными параметрами (другими словами, задаются по возвращении из вызова *Netbios*). Один важный совет:

всегда обнуляйте структуру *NCB* до вызова *Netbios*. В следующем списке описано назначение каждого поля.

**III *ncb\_command*** — указывает выполняемую команду NetBIOS. Многие команды могут выполняться синхронно или асинхронно поразрядным логическим сложением флага *ASYNCH* (0x80) и команды.

**II *ncbjretcode*** — определяет код возврата для данной операции. Функция присваивает этому полю значение *NRC\_PENDING*, когда происходит асинхронная операция.

- ***neb\_Isn*** — определяет номер локального сеанса, уникально идентифицирующий его в текущем окружении. Функция возвращает новый номер сеанса после успешной команды *NCBCALL* или *NCBLISTEN*.

**III *ncb\_num*** — указывает номер локального сетевого имени. Новый номер возвращается для каждого вызова команды *NCBADDNAME* или *NCBADDGRNAME*. Используйте корректный номер со всеми дейтаграммными командами.

**III *ncbbuffer*** — указывает на буфер данных. Для команд, которые отправляют данные, этот буфер содержит отправляемые данные; для команд, которые получают данные, — данные, возвращаемые функцией *Netbios*. Для других команд, типа *NCBENUM*, буфер будет предопределенной структурой *LANAJ.NUM*.

**III *ncb\_length*** — указывает длину буфера в байтах. Для команд приема *Netbios* присваивает этому полю значение, равное количеству полученных байтов. Если буфер недостаточно велик, *Netbios* возвращает ошибку *NRC\_BUFLen*.

**III *ncbcallnatne*** — указывает имя удаленного приложения.

- ***ncb\_name*** — указывает имя, под которым известно приложение.
- ***ncbrto*** — указывает время ожидания (тайм-аут) для операций приема. Значение определено в 500-миллисекундных единицах (0 — означает нулевое время ожидания) и задано для команд *NCBCALL* и *NCBLISTEN*, что влияет на последующие команды *NCBRCV*.
- ***ncbsto*** — указывает время ожидания для операций отправки. Значение определяется в 500-миллисекундных единицах (0 — означает нулевое время ожидания) и задано для команд *NCBCALL* и *NCBLISTEN*, что влияет на последующие команды *NCBSEND* и *NCBCHAINSEND*.

**III *ncb\_post*** — указывает адрес процедуры, которую надо вызвать по завершении асинхронной команды. Функция определена как

```
void CALLBACK PostRoutine(PNCB pncb);
```

*mdepncb* указывает на блок сетевого управления завершенной команды.

- ***ncb\_lana\_num*** — указывает номер LANA для выполнения команды.
- ***ncbcmcdplt*** — определяет код возврата для операции. *Netbios* присваивает этому полю значение *NRCJPENDING*, когда происходит асинхронная операция.
- ***ncb\_reserve*** — зарезервировано, должно быть равно 0.

- ***neb\_event*** — указывает описатель объекта события Windows в свободном (nonsignaled) состоянии. Когда асинхронная команда завершается, событие переходит в занятое (signaled) состояние. Следует использовать только ручной сброс событий. Это поле должно быть равно 0, если в поле ***neb\_command*** не задан флаг ***ASYNCH*** или если значение ноля ***nebjost*** от-  
лично от 0. Иначе ***Netbios*** возвращает ошибку ***NRCJLLCMD***.

## Синхронный и асинхронный вызов

Вы можете вызвать функцию ***Netbios*** синхронно или асинхронно. Все команды NetBIOS синхронны — это означает, что вызов ***Netbios*** блокируется, пока команда не завершит работу. При вызове команды ***NCBLISTEN*** запрос к ***Netbios*** не возвращается, пока клиент не установит соединение или не произойдет ошибка.

Для асинхронного вызова выполните логическую операцию OR над командой NetBIOS и флагом ***ASYNCH***. При использовании флага ***ASYNCH*** необходимо определить вызываемую после выполнения команды процедуру в поле ***ncb\_post***, либо описатель события в поле ***nebjvent***. Когда асинхронная команда выполнена, ***Netbios*** возвращает значение ***NRC\_GOODRET*** (0x00), но полю ***ncb\_cmd\_cplt*** присваивается значение ***NRC\_PENDING*** (0xFF). Кроме того, функция ***Netbios*** задает полю ***ncb\_cmd\_cplt*** структуры ***NCB*** значение ***NRC\_PENDING***, пока команда не завершится. По завершении команды полю ***ncb\_cmd\_cplt*** присваивается значение, возвращенное командой. После своего завершения ***Netbios*** также присваивает полю ***ncb\_retcode*** свой код возврата.

## Типичные процедуры NetBIOS

В этом разделе мы исследуем типичное серверное приложение NetBIOS. Сначала изучим сервер, так как его конструкция диктует действия клиента. Поскольку большинство серверов предназначено для обслуживания нескольких клиентов одновременно, асинхронная модель NetBIOS подойдет лучше всего. Мы обсудим примеры серверов, использующих как асинхронные процедуры обратного вызова, так и модель событий. Однако для начала представим исходный текст, который реализует некоторые обычные функции, необходимые большинству прикладных программ NetBIOS. Листинг 1-1 взят из файла ***Nbcommon.c***, который вы найдете на прилагаемом компакт-диске в папке ***\Examples\Chapter01\Common***, Функции из этого файла используются во многих примерах книги.

### Листинг 1-1. Типичные процедуры NetBIOS (***Nbcommon.c***)

```
// Nbcommon.c

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

см. след. стр.



Листинг 1 - 1. *(продолжение)*

```

«include «nbcommon.h"

//
// Перечисляются все номера LANA
//
int LanaEnum(LANA_ENUM «lenum)
{
    NCB          ncb,
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (PUCHAR)lenum;
    ncb.ncb_length = sizeof(LANA_ENUM);
    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR Netbios: NCBENUM Xd\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    >
    return NRC_GOODRET;

// Оброс всех сведений о LANA, перечисленных в структуре LANA_ENUM,
// а также настройка среды NetBIOS (максимальное количество сеансов,
// максимальный размер таблицы имен),и использование первого NetBIOS-имени.
//
int ResetAll(LANA_ENUM *lenum, UCHAR ucMaxSession,
             UCHAR ucMaxName, BOOL bFirstName)
{
    NCB          ncb;
    int          i;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_callname[0] = ucMaxSession;
    ncb.ncb_callname[2] = ucMaxName;
    ncb.ncb_callname[3] = (UCHAR)bFirstName;

    for(i = 0; i < lenum->length,

        ncb.ncb_lana_num = lenum->lana[i];
        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR Netbios NCBRESET[Xd]: Xd\n"
                   ncb.ncb_lana_num, ncb.ncb_retcode);
            return ncb.ncb_retcode;

```

## Листинг 1-1. (продолжение)

```

    return NRC_GOODRET,

// Добавление указанного имени данному LANA. Возвращает номер
// для зарегистрированного имени
//
int AddName(int lana, char «name, int *num)
{
    NCB          ncb,

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDNAME,
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR Netbios NCBADDNAME[lana=Xd;name=Xs]: Xd\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET,

// Добавление указанного группового имени NetBIOS данному LANA
// Возвращение номера добавленного имени.
//
int AddGroupName(int lana, char «name, int *num)
{
    NCB          ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDGRNAME;
    ncb.ncb_lana_num = lana,
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.nob_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR Netbios NCBADDGRNAME[lana=Xd;name=Xs]: Xd\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num,

```

см. след. стр.

**Листинг 1-1.**      *(продолжение)*

```

return NRC_GOODRET;

// Удаление данного имени NetBIOS из таблицы имен,
// связанной с номером LANa
//
int DelName(int lana, char *name)
{
    NCB                neb;

    ZeroMemory(&neb, sizeof(NCB));
    neb.ncb_command = NCBDELNAME;
    neb.ncb_lana_num = lana;
    memset(neb.ncb_name, ' ', NCBNAMESZ);
    strncpy(neb.ncb_name, name, strlen(name));

    if (Netbios(Snbc) != NRC_GOODRET)
    {
        printf("ERROR: Netbios NCBADDNAME[lana=%d; name=%s]: %d\n",
            lana, name, neb.ncb_retcode);
        return neb.ncb_retcode;
    }
    return NRC_GOODRET;

// Отправка len байт из буфера данных по указанному сеансу (lsn)
// и номеру lana
//
int Send(int lana, int lsn, char <data, DWORD len)
{
    NCB                neb,
    int                retcode;

    ZeroMemory(&neb, sizeof(NCB));
    neb.ncb_command = NCBSEND,
    neb.ncb_buffer = (PUCHAR)data,
    neb.ncb_length = len;
    neb.ncb_lana_num = lana;
    neb.ncb_lsn = lsn,

    retcode = Netbios(&neb);

    return retcode;

// Прием не более len байт в буфер данных по указанному сеансу

```

## Листинг 1-1. (продолжение)

Ц (lsn) и номеру lana

```

//
int Recv(int lana, int lsn, char «buffer, DWORD *len)
{
    NCB                neb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRECV;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = *len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    if (Netbios(&ncb) != NRC.GOODRET)
    {
        *len = -1;
        return ncb.ncb_retcode;
    }
    *len = ncb.ncb_length;

    return NRC_GOODRET;
}
//
// Прекращение указанного сеанса на данном номере lana
//
int Hangup(int lana, int lsn)
{
    NCB                neb;
    int                retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    neb.ncb..command = NCBHANGUP;
    neb.ncb..lsn = lsn
    neb.ncb_lana_num = lana;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Отмена данной асинхронной команды, указанной в параметре-структуре NCB
//
int Cancel(PNCB pncb)
{
    NCB                neb;

    ZeroMemory(&ncb, sizeof(NCB));

```

см.след.стр.

## Листинг 1-1.    (продолжение)

```

ncb.ncb_command = NCBCANCEL;
ncb.ncb_buffer = (PUCHAR)pncb;
neb.ncb_lana_num = pncb->ncb_lana_num;

if (Netbios(&ncb) != NRC_GOODRET)
{
    printf("ERROR: NetBIOS: NCBCANCEL: Xd\n", ncb.nob_retcode);
    return ncb.ncb_retcode;
}
return NRC_GOODRET;

// Форматирование указанного имени NetBIOS, чтобы оно было пригодно для печати.
// Непечатаемые символы заменяются точками.
// Буфер outname содержит возвращенную строку, длина которой - минимум
// NCBNAMSZ + 1 символов.
//
int FormatNetbiosName(char «nname, char «outname)
{
    int i;

    strncpy(outname, nname, NCBNAMSZ);
    outname[NCBNAMSZ - 1] = 0;
    for(i = 0; i < NCBNAMSZ - 1; i++)
    {
        // Если символ непечатаемый, он заменяется точкой.
        if (!(outname[i] >= 32) && (outname[i] <= 126))
            outname[i] = '.';
    }
    return NRC_GOODRET;
}

```

Первая из типичных процедур, приведенных в файле *Nbcommon.c* — *LanaEnum*, самая распространенная: ее используют почти все приложения NetBIOS. Эта функция перечисляет доступные номера LANA в данной системе. Функция обнуляет структуру *NCB*, присваивает полю *nebjoommand* значение *NCBENUM*, полю *ncb\_buffer* — структуру *LANA\_ENUM* и полю *neblength* — значение размера структуры *LANAJ5NUM*. При правильной инициализации структуры *NCB* единственное действие, которое должна предпринять функция *LanaEnum*, чтобы вызвать команду *NCBENUM* — вызвать функцию *Netbios*. Как видите, выполнить команду NetBIOS несложно. В случае синхронных команд возвращаемое *Netbios* значение сообщит, успешно ли выполнена эта команда. Константа *NRC\_GOODRET* всегда соответствует успеху.

В случае успешного вызова NetBIOS в предоставленную структуру *LANAJ5NUM* будут записаны количество номеров LANA на текущем компьютере, а также фактические номера LANA. Структура *LANA\_ENUM* определена следующим образом:

```
typedef struct LANA_ENUM .
{
    UCHAR    length;
    UCHAR    lana[MAX_LANA + 1];
} LANA_ENUM, *PLANA_ENUM;
```

Поле *length* указывает, сколько номеров LANA имеет локальный компьютер, поле *lana* — массив фактических номеров LANA, значение *length* — сколько элементов массива *lana* будет заполнено номерами LANA.

Следующая функция — *ResetAll*, также используется во всех приложениях NetBIOS. Хорошо написанная программа NetBIOS должна обнулить каждый номер LANA, который планирует применить. Как только вы получаете структуру *LANA\_ENUM* с номерами LANA от *LanaEnum*, можете сбросить их, вызвав команду *NCBRESET* для каждого номера LANA в структуре. Это именно то, что делает *ResetAll*; первый параметр функции — структура *LANA\_ENUM*. Для сброса требуется, только чтобы функция присвоила полю *neb\_command* значение *NCBRESET*, а полю *ncbjanajium* — номер LANA, который требуется сбросить. Хотя некоторые платформы, типа Windows 95, не требуют инициализировать каждый используемый номер LANA, лучше все-таки это делать. Windows NT требует инициализировать каждый номер LANA до его использования, иначе любые другие запросы *KNetbios* вернут ошибку 52 (*NRCJENVNOTDEF*).

Кроме того, при сбросе номера LANA вы можете задать определенные параметры среды NetBIOS через символьные поля *ncbjzallname*. Другие параметры *ResetAll* соответствуют этим параметрам окружения. Функция использует параметр *ucMaxSession*, чтобы присвоить значение нулевому символу *ncb\_callname*, который задает максимальное количество сеансов. Обычно операционная система задает значение по умолчанию меньше максимума. Например, в Windows NT 4 по умолчанию допустимо 64 параллельных сеанса. *ResetAll* присваивает символу 2 поля *ncb\_callname* (определяющему максимальное число имен NetBIOS, которые могут быть добавлены к каждому номеру LANA) значение параметра *ucMaxName*. Опять же, ОС определяет стандартный максимум. Наконец, *ResetAll* присваивает символу 3, используемому для клиентов NetBIOS, значение своего параметра *bFirstName*. Присваивая этому параметру *TRUE*, клиент использует имя компьютера как имя его процесса NetBIOS. В результате клиент может соединяться с сервером и отправлять данные, не принимая никаких входящих соединений. Этот параметр ускоряет инициализацию (добавление имени NetBIOS в локальную таблицу требует времени).

Операцию добавления имени в локальную таблицу имен выполняет функция *AddName*. Параметры: добавляемое имя и номер LANA, к которому оно добавляется. Помните, что таблица имен у каждого LANA своя, и если вы хотите, чтобы ваше приложение обслуживало соединения на любом доступном номере LANA, добавьте имя процесса к каждому LANA. Команда для добавления уникального имени — *NCBADDNAME*. Другие обязательные поля — номер LANA, для которого добавляется имя, и добавляемое имя, которое должно быть скопировано в поле *ncbjname*. *AddName* сначала заполняет буфер *ncbjname* пробелами и предполагает, что параметр *name* указывает на строку с симво-

лом /O в конце. После успешного добавления имени *Netbios* возвращает в поле *ncbjium* номер добавленного имени NetBIOS. Это значение используется для дейтаграмм (подробно мы обсудим их далее), чтобы идентифицировать исходный процесс NetBIOS. Наиболее типичная ошибка, с которой сталкиваются при добавлении уникального имени — *NRCJDUPNAME*, происходит, когда имя уже используется другим процессом в сети.

*AddGroupName* работает так же, как *AddName*, за одним исключением: *AddGroupName* запускает команду *NCBADDGRNAME* и никогда не вызывает ошибку *NRCJDUPNAME*.

Функция *DelName* удаляет имя NetBIOS из таблицы имен. Ей требуются только номер LANA, для которого вы хотите удалить имя, и само имя.

Следующие две функции в листинге 1-1 — *Send* и *Recv*, используются для отправки и получения данных в сеансе связи. Эти функции почти идентичны за исключением значения поля *neb\_command*. ему присваивается *NCBSEND*, либо *NCBRCV*. Обязательные параметры команды: номер LANA, по которому будут отправлены данные, и номер сеанса. Успешная команда *NCBCALL* или *NCBLISTEN* возвращает номер сеанса. Клиенты используют команду *NCBCALL* для соединения с известной службой, а серверы — команду *NCBLISTEN*, чтобы ждать входящих клиентских соединений. Когда какая-либо из этих команд успешно выполняется, интерфейс NetBIOS устанавливает сеанс с уникальным целым идентификатором.

*Send* и *Recv* также требуют параметров, которые проецируются в поля *ncbjouffer* (буфера) и *ncbjlength* (длины). При отправке данных поле *ncb\_buffer* указывает на содержащий эти данные буфер. Поле длины задает количество отправляемых символов в буфере. При получении данных поле буфера указывает на блок памяти, в который копируются входящие данные. Поле длины содержит размер блока памяти. Когда функция *Netbios* завершает работу, она записывает в поле длины количество успешно полученных байтов.

Важный аспект отправки данных через сеанс: вызов функции *Send* не осуществим, пока получатель не вызовет функцию *Recv*. То есть если отправитель выдает большое количество данных, а получатель не читает их, используются значительные ресурсы для локальной буферизации данных. Поэтому лучше вызывать немного команд *NCBSEND* или *NCBCHAINSEND* одновременно. Для решения этой проблемы используйте *Netbios-команды* *NCBSENDNA* и *NCBCHAINSENDNA*. В этом случае отправлять данные можно без подтверждения получателя.

Последние две функции в конце листинга 1-1 — *Hangup* и *Cancel*, предназначены для закрытия установленных сеансов или отмены невыполненной команды. Вызов команды NetBIOS *NCBHANGUP* позволит корректно завершить сеанс. При этом все невыполненные запросы на получение данных завершаются и возвращают ошибку закрытого сеанса — *NRC\_SCLOSED* (0x0A). Если какие-либо команды отправки данных не выполнены, команда завершения связи блокируется вплоть до их выполнения. Эта задержка происходит независимо от того, передает ли команда данные или ожидает, когда удаленная сторона запросит прием данных.

## Сервер сеансов: модель асинхронного обратного вызова

Займемся сервером, который будет слушать входящие клиентские соединения. Это простой эхо-сервер, отправляющий обратно любые данные, полученные от клиента. Листинг 1-2 содержит код сервера, использующего асинхронные функции обратного вызова. Код взят из файла Cbnbsvr.c, на прилагаемом компакт-диске он находится в папке /Examples/Chapter01/Server. В функции *main* сначала перечисляются доступные номера LANA с помощью *LanaEnum*, а затем — сбрасывается каждый номер LANA с помощью *ResetAll*. Помните, что эти два шага обычно требуются от всех приложений NetBIOS.

Листинг 1-2. Сервер асинхронного обратного вызова (Cbnbsvr.c)

```
// Cbnbsvr.c

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

#include "..\Common\nbcommon.h"

#define MAX.BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

DWORD WINAPI ClientThread(PVOID lpParam);

// функция: ListenCallback
//
// Описание:
// Эта функция вызывается, когда завершается асинхронное прослушивание.
// Если не происходит никакой ошибки, создает поток, чтобы работать клиентом.
// Также асинхронно дается команда слушать другие клиентские соединения.
//
void CALLBACK ListenCallback(PNCB pncb)
{
    HANDLE      hThread;
    DWORD       dwThreadId;

    if (pncb->ncb_retcode != NRC_GOODRET)
    {
        printf("ERROR: ListenCallback: Xd\n", pncb->ncb_retcode);
        return;
    }
    Listen(pncb->ncb_lana_num, SERVER_NAME);

    hThread = CreateThread(NULL, 0, ClientThread, (PVOID)pncb, 0,
        &dwThreadId);
    if (hThread == NULL)
```

ы АМАЖ по  
, (кшк до. см. след. стр.



Листинг 1-2. *(продолжение)*

```

{
    printf("ERROR: CreateThread: Xd\n", GetLastError());
    return;
}
CloseHandle(hThread);

return;

// Функция: ClientThread
//
// Описание:
// Клиентский поток блокирует получение данных от клиентов и
// просто посылает их назад. Это непрерывный цикл, выполняющийся
// пока не будет закрыт сеанс или не произойдет ошибка. Если
// чтение или запись дадут ошибку NRC_SCLOSED, сеанс
// корректно завершается и происходит выход из цикла.
//
DWORD WINAPI ClientThread(PVOID lpParam)

    PNCB        pncb = (PNCB)lpParam;
    NCB          ncb;
    char         szRecvBuff[MAX_BUFFER];
    DWORD        dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;
    char         szClientName[NCBNAMSZ+1];

    FormatNetbiosName(pncb->ncb_callname, szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=Xd]: 'XsAn", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client 'Xs' on LANA Xd disconnected\n", szClientName,
        pncb->ncb_lana_num);

```

**Листинг 1-2.** *(продолжение)*

```

if (dwRetVal != NRC_CLOSED)
{
    // Возникает какая-то другая ошибка; соединение разрывается
    //
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = pncb->ncb_lsn;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    <
        printf("ERROR: Netbios: NCBHANGUP: %d\n", ncb.ncb_retcode);
        dwRetVal = ncb.ncb_retcode;
    >
    GlobalFree(pncb);
    return dwRetVal;
}
GlobalFree(pncb);
return NRC_GOODRET;

// Функция: Listen
//
// Описание:
// Иницируется асинхронное прослушивание с помощью функции обратного вызова.
// Создается структура NCB для использования обратным вызовом (поскольку она
// должна быть видима глобально).
//
int Listen(int lana, char *name)
{
    PNCB          pneb = NULL;

    pneb = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, sizeof(NCB));
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    pncb->ncb_post = ListenCallback;
    //
    // Это имя, с которым клиенты будут соединяться
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // '*' означает, что мы примем клиентское соединение от любого клиента.
    // Задавая здесь конкретное имя, мы разрешаем соединение только с
    // указанным клиентом.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);

```

*см.след.стр.*

Листинг 1-2. (продолжение')

```
pncb->ncb_callname[0] = '.';

if (Netbios(pncb) != NRC_GOODRET)
{
    printf("ERROR: Netbios: NOBLISTEN: Xd\n", pncb->ncb_retcode);
    return pncb->ncb_retcode;
}
return NRC_GOODPET;
}

<n. m

И
II Функция: main , "n/bl :ЧШ>"
//
// Описание:
// Инициализирует интерфейс NetBIOS, выделяет некоторые ресурсы, добавляет
// имя сервера к каждому номеру LANa и дает асинхронную команду NCBLISTEN на
// каждый номер LANa с соответствующим обратным вызовом. Затем ждет входящих
// клиентских соединений, порождая рабочий поток, чтобы
// работать с ними. Главный поток просто ждет, пока серверные
// потоки работают с клиентскими запросами. Вы навряд ли будете делать это в {
II настоящем приложении: Этот пример приведен только для иллюстрации.
//
int main(int argc, char **argv) , <d
< W
    LANa.ENUM lenum; d
    int i, ,\
        num; d
        A
// Перечисляются и инициализируются все номера LANa d
// I1 *ni
if (LanaEnum(&lenum) != NRC_GOODRET) }
    return 1; n
if (ResetAlk&lenum, 254, 254, FALSE) != NRC.GOODRET)
    return 1;
//
// Каждому номеру LANa добавляется имя сервера и дается команда слушать
//
for(i=0; i< lenum.length;
    AddName(lenum.lana[i], SERVER_NAME, inura);
    Listen(lenum.lana[i], SERVER_NAME);
> ,i
    \
while (1) \
< \
    Sleep(5000); \
> S
```

Далее функция *main* добавляет имя вашего процесса к каждому номеру LANА, на котором вы хотите принимать соединения. Сервер добавляет имя своего процесса — TEST-SERVER-1, к каждому номеру LANА. Это имя (дополненное пробелами, конечно) клиенты будут использовать для соединения с сервером. При попытке установить или принять соединение важен каждый символ в имени NetBIOS. Большинство проблем, с которыми сталкиваются при программировании клиентов и серверов NetBIOS, связаны с несоответствием имен. Будьте последовательны, дополняя имена пробелами или каким-либо другим символом. Пробелы — наиболее популярные символы-заполнители, так как их удобно перечислять и распечатывать.

Последний и наиболее важный шаг для сервера — асинхронно выдать ряд команд *NCBLISTEN*. Функция *Listen* сначала выделяет структуру *NCB*. При использовании асинхронных вызовов NetBIOS, переданная вами структура *NCB* должна сохраняться с момента вызова до его завершения. Для этого требуется динамически выделять каждую структуру *NCB* перед выдачей команды или поддерживать глобальный пул структур *NCB*, чтобы использовать его в асинхронных запросах. Для команды *NCBLISTEN* задайте номер LANА, к которому должен обращаться вызов. Заметьте, что код в листинге 1-1 выполняет логическую операцию OR над командами *NCBLISTEN* и *ASYNCH*. При указании команды *ASYNCH* поле *ncb\_post* или *ncb\_event* не должно равняться 0, иначе вызов *Netbios* даст ошибку *NRCJLLCMD*.

В листинге 1-2 функция *Listen* присваивает полю *ncb\_post* функцию обратного вызова *ListenCallback*. Затем функция *Listen* присваивает полю *ncbname* имя процесса сервера. С этим именем будут соединяться клиенты. Функция также записывает в первый символ поля *ncb\_callname* астериск (\*). Это означает, что сервер примет соединение от любого клиента. С другой стороны, вы можете поместить в поле *ncbcallname* конкретное имя — тогда только зарегистрированный под этим именем клиент сможет соединиться с сервером. Наконец, функция *Listen* обращается к *Netbios*. Запрос завершается немедленно, и функция *Netbios* до завершения команды присваивает полю *ncbjzmdjzplt* переданной структуры *NCB* значение *NRC\_PENDING* (0xFF).

После того как *main* инициализируется и даст команду *NCBLISTEN* каждому номеру LANА, поток *main* переходит в бесконечный цикл.

**ПРИМЕЧАНИЕ** Так как этот сервер — только пример, схема его работы очень проста. При разработке собственных серверов NetBIOS вы можете написать другую обработку в главном цикле или выдать в цикле *main* синхронную команду *NCBLISTEN* одному из LANА.

Функция обратного вызова выполняется, только когда входящее соединение принято на номере LANА. Когда команда *NCBLISTEN* принимает соединение, она вызывает эту функцию в поле *ncb\_post* с исходной структурой *NCB* в качестве параметра. Полю *ncb\_retcode* присваивается значение кода возврата. Всегда проверяйте это значение, чтобы увидеть, последовало ли клиентское соединение. В случае успешного соединения поле *ncb\_retcode* будет равно *NRCJSOODRET* (0x00).

Если соединение успешно, асинхронно выдайте другую команду *NCBLISTEN* тому же самому номеру *LANA*. Это необходимо потому, что как только исходное прослушивание завершится успехом, сервер прекратит слушать клиентские соединения на этом *LANA*, пока не будет дана другая команда *NCBLISTEN*. Таким образом, чтобы упростить доступ к вашим серверам, вызовите несколько команд *NCBLISTEN* для одного *LANA* — тогда будет можно одновременно принимать соединения от множества клиентов. Наконец, функция обратного вызова создает поток, который будет обслуживать клиента. В нашем примере данный поток просто работает в цикле и вызывает команду блокирующего чтения (*NCBRCV*), а затем — блокирующей передачи (*NCBSEND*). В примере реализован эхо-сервер, читающий сообщения клиентов и отправляющий их обратно. Клиентский поток входит в цикл, пока клиент не прервет соединение, после чего клиентский поток дает команду *NCBHANGUP*, чтобы закрыть соединение со своей стороны. С этого момента клиентский поток освобождает структуру *NCB* и завершается.

Для сеансов данные буферизируются нижележащими протоколами, так что наличие невыполненных запросов на прием необязательно. После того, как дана команда приема, функция *Netbios* немедленно передает доступные данные в предоставленный буфер и вызов возвращается. Если доступных данных нет, вызов приема блокируется, пока не появятся доступные данные или не прервется сеанс. То же верно для команды отправления: если сетевой стек способен отправить данные немедленно по проводу или буферизовать их в стеке для дальнейшей передачи, вызов возвращается немедленно. Если у системы нет буферного пространства, чтобы немедленно отправить данные, запрос на отправку блокируется, пока буфер не освободится.

Для решения этой проблемы можно задействовать команду *ASYNCH* при передаче и приеме. Буфер, предоставленный для асинхронной передачи и приема, должен быть виден за пределами вызывающей процедуры. Еще один способ обойти блокирование заключается в использовании полей *ncb\_sto* и *ncbjrto*. Поле *ncb\_sto* задает тайм-аут отправления. Указывая ненулевое значение в 500-миллисекундных единицах, вы можете задать максимальную продолжительность блокирования отправки перед возвратом. Если время команды истекает, данные не посылаются. То же верно для времени ожидания приема: если данные не получены за указанное время, вызов завершается без передачи данных в буфер.

## Сервер сеансов: модель асинхронных событий

В листинге 1-3 приведен код эхо-сервера, отличный от кода в листинге 1-2. Здесь в качестве механизма оповещения о завершении используются события Win32. Модель событий похожа на модель обратного вызова, единственное отличие: в модели обратного вызова система выполняет ваш код, когда асинхронная операция завершается, а в модели событий — приложение должно убедиться в завершении операции, проверяя состояние события. Поскольку речь идет о стандартных событиях Win32, вы можете использовать любую из доступных процедур синхронизации, например *WaitForSingleEvent* или *WaitForMultipleEvents*. Модель событий более эффективна, так как застав-

ляет программиста структурировать программу, чтобы сознательно проверять завершение.

Наш сервер модели событий начинает работу с того же, что и сервер обратного вызова.

1. Перечисляет номера LANA.
2. Сбрасывает все номера LANA.
3. Добавляет имя сервера к каждому номеру LANA.
4. Иницирует прослушивание на каждом LANA.

Единственное различие — нужно отслеживать все невыполненные команды прослушивания, чтобы непременно сопоставить завершение события с соответствующими блоками *NCB*, инициализирующими конкретную команду. Код в листинге 1-3 выделяет массив структур *NCB*, размер которого соответствует количеству номеров LANA (поскольку требуется выполнять команду *NCBLISTEN* для каждого номера). Дополнительно код создает событие для каждой структуры *NCB*, чтобы оповещать о завершении команды. Функция *Listen* использует одну из структур *NCB* из массива в качестве параметра.

Листинг 1-3. Сервер на основе асинхронных событий (Evnbsvr.c)

```
// Evnbsvr.c

<include <windows.h>
<include <stdio.h>
<include <stdlib.h>

<include ".. \Common\nbcommon.h"

<define MAX_SESSIONS    254
<define MAX_NAMES       254

<define MAX_BUFFER      2048
<define SERVER_NAME     "TEST-SERVER-1"

NCB    *g_Clients=NULL;          // Global NCB structure for clients

// функция: ClientThread
//
// Описание:
// Этот поток берет структуру NCB из сеанса соединения
// и ждет входящих данных, которые затем посылает обратно
// клиентам, пока сеанс не будет закрыт.
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB    pncb = (PNCB)lpParam;
    NCB      neb;
    char     szRecvBuff[MAX_BUFFER],
```

см. след. стр.

### Листинг 1-3. (продолжение)

```

        szClientName[NCBNAHSZ + 1];
DWORD    dwBufferLen = MAX_BUFFER,
        dwRetVal = NRC_GOODRET;

// Отправка и прием сообщений, пока сеанс не будет закрыт
//
FormatNetbiosName(pncb->ncb_callname, szClientName);
while (1)
{
    dwBufferLen = MAX_BUFFER;
    dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
        szRecvBuff, idwBufferLen);
    if (dwRetVal != NRC_GOODRET)
        break;

    szRecvBuff[dwBufferLen] = 0;
    printf("READ [LANA=Xd]: 'Xs'\n", pncb->ncb_lana_num,
        szRecvBuff);

    dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
        szRecvBuff, dwBufferLen);
    if (dwRetVal != NRC_GOODRET)
        break;
}
prmtf("Client 'Xs' on LANA Xd disconnected\n", szClientName,
    pncb->ncb_lana_num);

//
// Если в ходе чтения или записи выдается ошибка NRC.SCLOSED,
// то все в порядке; иначе возникла некоторая другая ошибка, так что
// соединение разрывается с этой стороны.
//
if (dwRetVal != NRC_SCLOSED)
{
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = pncb->ncb_lsn;
    ncb.ncb_lana_num = pncb->ncb_lana_num;
    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBHANGUP: Xd\n",
            neb.ncb_retcode);
        GlobalFree(pncb);
        dwRetVal = neb.ncb_retcode;

        // Передаваемая структура NCB выделяется динамически, поэтому
        // ее следует вначале удалить

```

### Листинг 1-3. (продолжение)

```
GlobalFree(pncb);
return NRC_GOODRET;

// Функция: Listen
//
// Описание:
// Дается команда асинхронно слушать на указанном LANA.
// В переданной структуре NCB полю ncb_event уже
// присвоено значение действительного описателя события Windows.
//
int Listen(PNCB pncb, int lana, char «name»)
{
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    //
    // Это имя, с которым будут соединяться клиенты
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // '*' означает, что мы примем клиентское соединение от любого клиента.
    // Задавая здесь конкретное имя, мы разрешаем соединение только с
    // указанным клиентом.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';
    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: Xd\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;

// Функция: main
//
// Описание:
// Инициализирует интерфейс NetBIOS, выделяет некоторые ресурсы, и
// дает асинхронную команду слушать каждому LANA, используя события. Ждет
// пока событие не сработает, а затем обрабатывает клиентское соединение.
//
int main(int argc, char **argv)
{
    PNCB          pncb=NULL;
```

см. след. стр.



Листинг 1 -3. (продолжение)

```

HANDLE      hArray[64],
            hThread;
DWORD       dwHandleCount=0,
            dwRet,
            dwThreadId;

int         i,
            num;
LANA_ENUM   lenum;

// Перечисление и сброс всех номеров LANA
//
if (LanaEnum(&lenum) != NRC_GOODRET)
    return 1;
if (ResetAlk&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
    FALSE) != NRC_GOODRET)
    return 1;
//
// Выделение массива структур NCB (по одной для каждого номера LANA)
//
g_Clients = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(NCB) * lenum.length);
//
// Создание событий, добавление имени сервера каждому номеру LANA и начало
// асинхронного прослушивания на каждом LANA.
//
for(i = 0; i < lenum.length; i++)
{
    hArray[i] = g_Clients[i].ncb_event = CreateEvent(NULL, TRUE,
        FALSE, NULL);

    AddName(lenum.lana[i], SERVER_NAME, &num);
    Listen(&g_Clients[i], lenum.lana[i], SERVER.NAME);
}
while (1)
{
    // Ожидание подключения клиента
    //
    dwRet = WaitForMultipleObjectsdenum.length, hArray, FALSE,
        INFINITE);
    if (dwRet == WAIT_FAILED)
    {
        printf("ERROR: WaitForMultipleObjects: Xd\n",
            GetLastError());
        break;
    }
    // Проверка всех структур NCB для определения, достигла ли успеха более
    // чем одна структура. Если поле ncb_cmd_plt не содержит значение
    // NRC_PENDING, значит существует клиент, необходимо создать поток и
    // выделить ему новую структуру NCB.

```

### Листинг 1-3. (продолжение)

```
// Нам нужно многократно использовать исходную структуру
// NCB для других клиентских соединений.
//
for(i = 0; i < lenum.length; i++)
<
    if (g_Clients[i].ncb_cmd_cplt != NRC_PENDING)
    {
        pncb = (PNCB)GlobalAlloc(GMEM_FIXED, sizeof(NCB));
        memcpy(pncb, &g_Clients[i], sizeof(NCB));
        pncb->ncb_event = 0;

        hThread = CreateThread(NULL, 0, ClientThread,
            (LPVOID)pncb, 0, &dwThreadId);
        CloseHandle(hThread);
        //
        // Описатель сбрасывается, асинхронно начинается еще одно
        // прослушивание ResetEvent(hArray[i]);
        //
        Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);

        // Очистка
    }
    //
    for(i = 0; i < lenum.length; i
? <
~* DelName(lenum.lana[i], SERVER_NAME);
n11 CloseHandle(hArray[i]);
К. ,
is GlobalFree(g_Clients);

- return 0;
```

Первый цикл функции *main* получает доступные номера LANA и при этом добавляет имя сервера и отдает команду *NCBLJSTEN* для каждого LANA, а также формирует массив описателей событий. Затем вызывается функция *WaitForMultipleObjects*, которая блокируется, пока по меньшей мере один из описателей не перейдет в занятое состояние. Тогда *WaitForMultipleObjects* завершается, и код порождает поток для чтения входящих сообщений и отправки их обратно клиенту. Код копирует занятую структуру *NCB*, чтобы передать ее в клиентский поток. Это позволяет многократно использовать исходную структуру *NCB* для асинхронного вызова других команд *NCBLISTEN*, что можно сделать путем сброса события и повторного вызова функции *Listen* для этой структуры. Не обязательно копировать всю структуру — на самом деле вам нужны только локальный номер сеанса (*ncbjsri*) и номер LANA (*ncbjana[nctri]*). Впрочем, структура *NCB* — удобный контейнер для хранения обоих значений, чтобы потом передать их в один параметр потока. Кли-

ентский поток, используемый моделью событий, тот же, что и для модели обратного вызова, за исключением оператора *GlobalFree*

## Асинхронные стратегии сервера

При использовании обоих видов серверов существует вероятность что клиенту будет отказано в обслуживании. По завершении команды *NCBLJSTEN* происходит небольшая задержка, которая длится вплоть до вызова функции обратного вызова или пока событие не будет занято. Рассмотренные нами серверы не отдадут асинхронно еще одну команду *NCBLJSTEN* ранее, чем через несколько операторов. Например, если сервер обслуживал клиента на номере *LANA 2* и еще один клиент попытается подключиться прежде, чем сервер отдаст следующую команду *NCBLJSTEN* на том же номере *LANA*, клиент получит ошибку *NRC\_NOCALL (0x14)*. Это означает, что для данного имени еще не была асинхронно вызвана команда *NCBLJSTEN*. Чтобы избежать такой ситуации, сервер может асинхронно отдавать несколько команд *NCBLJSTEN* на каждом *LANA*.

Как видите, использовать асинхронные команды достаточно просто. Флаг *ASYNCH* может применяться почти к любой команде NetBIOS. Помните только, что структура *NCB*, которую вы передаете *Netbios*, должна быть видима глобально.

## Клиент сеанса NetBIOS

[

Клиент NetBIOS напоминает асинхронный сервер событий. Листинг 1-4 содержит пример кода для клиента. Клиент выполняет уже знакомые нам стандартные шаги инициализации имени добавляет свое имя к таблице имен каждого номера *LANA* и затем дает асинхронную команду соединения. Цикл *main* ждет, пока одно из событий не будет занято. В это время в цикле проверяется поле *ncb\_cmdj3ptt* каждой структуры *NCB*, соответствующей отданным командам подключения, по одной для каждого номера *LANA*. Если поле *ncb\_cmd\_cplt* равно *NRC\_PENDLNG* — код отменяет асинхронную команду, если команда завершается (соединение установлено) и данная *NCB* не соответствует оповещенной (определяется по значению, возвращаемому *WaitForMultipleObjects*) — соединение разрывается. Когда сервер слушает на каждом номере *LANA* на своей стороне и клиент пытается соединиться на каждом из его номеров *LANA*, может быть установлено несколько соединений. В этом случае код просто закрывает лишние соединения командой *NCBHANGUP* — нужна связь только по одному каналу. Попытки соединения с каждым номером *LANA* с обеих сторон практически гарантированно успешны.

### Листинг 1-4. Клиент на основе асинхронных событий (Nbclient.c)

```
// Nbclient.c
```

```
((include <windows.h>
<include <stdio.h>
((include <stdlib.h>
```

## Листинг 1-4. (продолжение)

```

#include    \Common\nbcommon.h

#define MAX_SESSIONS    254
#define MAX_NAMES       254

#define MAX_BUFFER       1024

char    szServerName[NCBNAMSZ];

//
// функция Connect
//
// Описание
// Установка асинхронного соединения с сервером на данном LANa
// В переданной структуре NCB полю ncb_event уже присвоено значение
// действительного описателя события Windows Просто
// заполните пробелы и сделайте вызов
//
int Connect(PNCB pncb, mt lana, char *server, char «client)
{
    pncb->ncb_command = NCBCALL | ASYNCH,
    pncb->ncb_lana_num = lana,

    memset(pncb->ncb_name,    , NCBNAMSZ),
    strncpy(pncb->ncb_name, client, strlen(client));

    memset(pncb->ncb_callname,    , NCBNAMSZ),
    strncpy(pncb->ncb_callname, server, strlen(server));

    if (Netbios(pncb) != NRC_GOODRET)
    {
        pnntf( ERROR Netbios NCBCONNECT Xd\n",
            pncb->ncb_retcode),
        return pncb->ncb_retcode,
    }
    return NRCJ3OODRET,

// функция main
//
// Описание
// Инициализирует интерфейс NetBIOS, распределяет некоторые ресурсы
// (описатели событий, буфер отправки и т п ) и дает команду
// NCBCALL для каждого номера LANa на указанном сервере Когда соединение
// создано, отменяет или разрывает любые неудавшиеся
// соединения Затем посылает/получает данные Наконец, выполняет очистку

```

## Листинг 1-4. {продолжение}

```
int. main(int argc, char **argv)
```

```

HANDLE      •hArray;
NCB          *pncb;
char         szSendBuff[MAX_BUFFER];
DWORD        dwBufferLen,
             dwRet,
             dwIndex,
             dwNum;
LANA_ENUM    lenum;
int          1;

if (argc != 3)

    printf("usage: nbclient CLIENT-NAME SERVER-NAME\n");
    return 1;
}
// Перечисление и сброс всех номеров LANA
//
if (LanaEnum(&lenum) != NRC_GOODRET)
    return 1;
if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
             FALSE) != NRC_GOODRET)
    return 1;
strcpy(szServerName, argv[2]);
//
// Выделение массива описателей для использования асинхронных событий.
// Выделение массива структур NCB. Нам нужен один описатель
// и одна структура NCB для каждого номера LANA.
//
hArray = (HANDLE *)GlobalAlloc(GMEM_FIXED,
    sizeof(HANDLE) * lenum.length);
pncb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(NCB) * lenum.length);
//
// Создание события и присвоение его соответствующей структуре NCB,
// начало асинхронного соединения (NCBCALL).
// Не забудьте добавить имя клиента к каждому номеру
// LANA, по которому он хочет соединиться.
for(i = 0; i < lenum.length; i++)
<
    hArray[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    pncb[i].ncb_event = hArray[i];
    AddName(lenum.lana[i], argv[1], &dwNum);
    Connect(&pncb[i], lenum.lana[i], szServerName, argv[1]);
}
// Ожидание успеха по меньшей мере одного соединения

```

## Листинг 1-4. (продолжение)

```

//
dwIndex = WaitForMultipleObjectsQenum.length, hArray, FALSE,
    INFINITE);
if (dwIndex == WAIT_FAILED)
{
    printf("ERROR: WaitForMultipleObjects: Xd\n",
        GetLastErrorO);

else

    // Если успешно более чем одно соединение, лишние
    // соединения разрываются. Мы будем использовать соединение, возвращенное
    // WaitForMultipleObjects, иначе если оно еще не установлено,
    // отменим его.
    //
    for(i = 0; i < lenum.length; i++)
    {
        if (i != dwIndex)
        {
            if (pncb[i].ncb_cmd_cplt == NRC.PENDING)
                Cancel(&pncb[i]);
            else
                Hangup(pncb[i].ncb_lana_num, pncb[i].ncb_lsn);

            printf("Connected on LANA; Xd\n", pncb[dwIndex].ncb_lana_num);
        }
    }

// Отправка и прием сообщений

for(i = 0; i < 20; i
{
    wsprintf(szSendBuff, "Test message X03d", i);
    dwRet = Send(pncb[dwIndex].ncb_lana_num,
        pncb[dwIndex].ncb_lsn, szSendBuff,
        strlen(szSendBuff));
    if (dwRet != NRCJ300DRET)
        break;
    dwBufferLen = MAX_BUFFER;
    dwRet = Recv(pncb[dwIndex].ncb_lana_num,
        pncb[dwIndex].ncb_lsn, szSendBuff, &dwBufferLen);
    if (dwRet != NRC_GOODRET)
        break;
    szSendBuff[dwBufferLen] = 0;
    printf("Read: %s\n", szSendBuff);
}
    Hangup(pncb[dwIndex].ncb_lana_num, pncb[dwIndex].ncb_lsn);
}
// Очистка

```

см. след. стр.

**Листинг 1-4.** (продолжение)

```
//
for(i = 0, i < lenum length,

    DelName(lenum lana[i], argv[1]),
    CloseHandle(hArray[i]),
)
GlobalFree(hArray),
GlobalFree(pncb),

return 0,
```

## Дейтаграммные операции

Дейтаграмма — это способ связи без установления логического соединения. Отправитель просто направляет каждый пакет по указанному имени NetBIOS. Целостность данных и порядок доставки пакетов не гарантируются.

Есть три способа отправить дейтаграмму. Первый — направить дейтаграмму на определенное (уникальное или групповое) имя. Это означает, что получить эту дейтаграмму может только процесс, зарегистрировавший имя приемника. Второй способ — отправить дейтаграмму на групповое имя: тогда получить сообщение смогут только процессы, зарегистрировавшие данное имя. Наконец, третий способ — ширококестельно разослать дейтаграмму по всей сети. Такую дейтаграмму сможет получить любой процесс на любой рабочей станции в локальной сети. Для отправки дейтаграммы на уникальное или групповое имя применяется команда *NCBDGSEND*, а для ширококестельного — *NCBDGSENDER*.

Отправить дейтаграмму с помощью любой из этих команд элементарно. Определите для поля *ncb\_num* значение номера имени, возвращенного командой *NCBADDNAME* или *NCBADDGRNAME*. Этот номер идентифицирует отправителя сообщения. Присвойте полю *ncbjmffer* значение адреса буфера, содержащего отправляемые данные, а полю *ncbjlength* — количество отправляемых байтов. Затем задайте для поля *ncbjname* значение номера LANA, по которому хотите передать дейтаграмму. Наконец, присвойте полю *ncbjcallname* значение NetBIOS-имени приемника. Это может быть уникальное или групповое имя. Чтобы ширококестельно послать дейтаграмму, выполните все описанные шаги, кроме последнего: так как сообщение получат все рабочие станции, присвоение значения полю *ncbjcallname* не требуется.

Конечно, в каждом из перечисленных сценариев отправления должна быть соответствующая команда получения дейтаграммы для фактического приема данных. Дейтаграммы не требуют установки соединения, если дейтаграмма достигает клиента, а у клиента нет уже ожидающей команды получения, данные теряются и клиент не может их восстановить (если сервер не отправит их снова). Это недостаток дейтаграммной связи. Впрочем, обмен дейтаграммами намного быстрее, чем сеансовая связь — не нужно проверять ошибки, устанавливать соединение и т. п.

**ПРИМЕЧАНИЕ** Невозможно дать команду асинхронного приема дейтаграмм, предназначенных для имени, которое зарегистрировано иным процессом, если только оба процесса не зарегистрировали групповое имя (тогда они могут получить одно и то же сообщение)

Код в листинге 1-5 содержит основные дейтаграммные функции. Вся отправка реализуется блокирующими вызовами как только команда выдана и данные отправлены, функция завершается и вам не грозит блокировка из-за переполнения буфера данными. Вызовы приема — асинхронные события, так как неизвестно на каком из номеров LANa будут получены данные. Код похож на код сервера сеансов, использующего события. Для каждого номера LANa код отдает асинхронную команду *NCBDGRCV* или *NCBDGRCVBC* и ждет, пока она не достигнет успеха. Затем проверяются все асинхронные команды, печатаются сообщения о тех, которые были успешны, и отменяются еще ожидающие команды. В примере есть функции, как для направленного, так и для широковещательного отправления и приема. Программа может быть скомпилирована в пример приложения, конфигурируемого для отправки или получения дейтаграмм. Несколько параметров командной строки позволяют пользователю указать количество отправляемых или принимаемых дейтаграмм, задержку между отправками, использование широковещательных, а не направленных дейтаграмм, получение дейтаграмм для любого имени и т. п.

```
// Nbdgram c
W §
«include <windows h> * i«
)in)t i cm cjieo.cmp.
```



Листинг 1-5. *(продолжение)*

```

«include <stdio.h>
«include <stdlib.h>

«include "..\Common\nbcommon.h"

«define MAX_SESSIONS      254
«define MAX_NAMES         254
«define MAX_DATAGRAM_SIZE 512

BOOL  bSender = FALSE,           // Отправка или прием дейтаграмм
      bRecvAny = FALSE,         // Прием для любого имени
      bUniqueName = TRUE,       // Зарегистрировать мое имя как уникальное?
      bBroadcast = FALSE,      // Использовать широковещательные
дейтаграммы?
      bOneLana = FALSE;         // Использовать все LANA или только один?
char  szLocalName[NCBNAMSZ + 1], // Локальное NetBIOS-имя
      szRecipientName[NCBNAMSZ + 1]; // NetBIOS-имя приема
DWORD dwNumDatagrams = 25,      // Количество отправляемых дейтаграмм
      dwOneLana,                // Если использовать один LANA, то какой?
      dwDelay = 0;              // Задержка между отправками дейтаграмм

// Функция: ValidateArgs

// Описание:
// Эта функция анализирует аргументы командной строки

// и устанавливает различные глобальные флаги, соответствующие выбору

void ValidateArgs(int argc, char **argv)

    int          i;

    for(i = 1; i < argc;

    {    if (strlen(argv[i]) < 2)
          continue;
          if ((argv[i][0] == '-' || (argv[i][0] == '\0' &
          {
              switch (tolower(argv[i][1]))

                  case 'n':           // Используется уникальное имя
                      bUniqueName = TRUE;
                      if (strlen(argv[i]) > 2)
                          strcpy(szLocalName, &argv[i][3]);
                      break;
                  case 'g':           // Используется групповое имя
                      bUniqueName = FALSE;
                      if (strlen(argv[i]) > 2)

```

Листинг 1-5.

```

        strcpy(szLocalName, &argv[i][3]);
        break;
    case 's':          // Отправка дейтаграмм
        bSender = TRUE;
        break;
if-    case 'c':          // Количество дейтаграмм для отправки или приема
        if (strlen(argv[i]) > 2)
            dwNumDatagrams = atoi(&argv[i][3]);
t/    break;
        case 'r':          // Имя получателя дейтаграмм
            if (strlen(argv[i]) > 2)
                strcpy(szRecipientName, &argv[i][3]);
            break;
        case 'b':          // Используется широковещательная рассылка
            bBroadcast = TRUE;
            break;
        case 'a':          // Прием дейтаграмм для любого имени
            bRecvAny = TRUE;
            break;
        case 'l':          // Работа только на этом номере LANA
            bOneLana = TRUE;
            if (strlen(argv[i]) > 2)
                dwOneLana = atoi(&argv[i][3]);
            break;
tf    case 'd':          // Задержка (в миллисекундах) между отправками
            if (strlen(argv[i]) > 2)
                dwDelay = atoi(&argv[i][3]);
            break;
        default:
            printf("usage: nbdgram ?\n");
            break;

        >
        return;

```

// Функция: DatagramSend

// Описание:

```

// Отправляет направленные дейтаграммы к указанному приемнику на
// заданном номере LANA от данного номера имени к соответствующему приемнику.
// Также указывается буфер данных и количество отправляемых байтов.

```

```

int DatagramSend(int lana, int num, char «recipient,
                 char «buffer, int buflen)

```

см. след. стр-

## Листинг 1-5. (продолжение)

```

NCB                neb;

ZeroMemory(&ncb, sizeof(NCB));
ncb.ncb_command = NCBDGSEND;
ncb.ncb_lana_num = lana;
neb.ncb_num = num;
ncb.ncb_buffer = (PUCHAR)buffer;
ncb.ncb_length = buflen;

memset(ncb.ncb_callname, ' ', NCBNAMSZ);
strcpy(ncb.ncb_callname, recipient, strlen(recipient));

if (Netbios(&ncb) != NRC_GOODRET)
{
    printf("Netbios: NCBDGSEND failed: Xd\n", neb.ncb_retcode);
    return ncb.ncb_retcode;
}
return NRC_GOODRET;

// Функция: DatagramSendBC
//
// Описание:
// Посылает широковещательную дейтаграмму по конкретному номеру LANA с
// данного номера имени. Также указаны буфер данных и количество
// отправляемых байт.
//
int DatagramSendBC(int lana, int num, char «buffer, int buflen)
{
    NCB                neb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.neb_command = NCBDGSENDBC;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSENDBC failed: Xd\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

```

Листинг 1-5. (продолжение)

II функция: DatagramRecv

```
//
// Описание:
// Получает дейтаграмму на данном номере LANA направленную по имени,
// представленному параметром num. Данные копируются в предоставленный буфер.
// Если hEvent не 0, вызов приема выполняется асинхронно
// с указанным описателем события. Если параметр num равен 0xFF, слушает
// дейтаграммы, предназначенные для любого имени NetBIOS,
// зарегистрированного процессом.
```

```
int DatagramRecv(PNCB pncb, int lana, int num, char «buffer,
                  int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDBGRECV | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDBGRECV;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer; , ^
    pncb->ncb_length = buflen;          " s , a j A i ! ч 1
    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbos: NCBDBGRECV failed: И\п", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;                  I8J*.
                                         , |
```

// функция: DatagramRecvBC

```
//
// Описание:
// Получает широковещательную дейтаграмму на данном номере LANA.
// Данные копируются в предоставленный буфер. Если hEvent не равно 0,
// вызов приема выполняется асинхронно с указанным
// описателем события.
//
int DatagramRecvBC(PNCB pncb, int lana, int num, char «buffer,
                   int buflen, HANDLE hEvent)
```

```
ZeroMemory(pncb, sizeof(NCB));
if (hEvent)
```

, "n/bt:to9ix\*1

см.след.смп.

**Листинг 1-5. (продолжение)**

```
{
    pncb->ncb_command = NCBDGRECVBC | ASYNCH;
    pncb->ncb_event = hEvent;
}
else
    pncb->ncb_comrnand = NCBDGRECVBC;
pncb->ncb_lana_num = lana;
pncb->ncb_num = min;
pncb->ncb_buffer = (PUCHAR)buffer;
pncb->ncb_length = buflen;

'if (Netbios(pncb) != NRC_GOODRET)
*{
    printf("Netbios: NCBDGRECVBC failed: Xd\n", pncb->ncb_retcode);
    return pncb->ncb_retcode;

return NRC_GOODRET;
```

II Функция: main

```
//
II Описание:
// Инициализирует интерфейс NetBIOS, выделяет ресурсы, а затем отправляет
// получает дейтаграммы согласно пользовательским параметрам
//
int main(int argc, char *»argv)
i
    LANA_ENUM lenum;
    int i, j;
    char ' " szMessage[MAX_DATAGRAM_SIZE],
        szSender[NCBNAMSZ + 1];
    DWORD «dwNulfi = NULL,
        dwBytesRead,
        dwErr;

    ValidateArgs(argc, argv);
    //
    // Перечисление и сброс номеров LANA
    //
    if ((dwErr = LanaEnum(&lenum)) != NRC_GOODRET)
    {
        printf("LanaEnum failed: Xd\n", dwErr);
        return 1;
    }
    if ((dwErr = ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
        (UCHAR)MAX_NAMES, FALSE)) != NRC_GOODRET)
    {
        printf("ResetAll failed: Xd\n", dwErr);
```

Листинг 1-5. (продолжение)

```

return 1;

// Этот буфер содержит номер добавленного к каждому номеру LANA
// NetBIOS-имени
//
dwNum = (DWORD .)GlobalAlloc(GMEH_FIXED | GMEM_ZEROINIT,
    sizeof(DWORD) * lenum.length);
if (dwNum == NULL)
{
    printf("out of memory\n");
    return 1;

// Если мы собираемся работать только на одном номере LANA, имя регистрируется
// только на этом номере LANA, иначе имя регистрируется на всех
// номерах LANA
//
if (bOneLana)                // \i
{
    if (bUniqueName)
        AddName(dwOneLana, szLocalName, &dwNum[0]);
    else
        AddGroupName(dwOneLana, szLocalName, &dwNum[0]);
}
else
{
    for(i = 0; i < lenum.length; i++)
    {
        if (bUniqueName)
            AddName(lenum.lana[i], szLocalName, &dwNum[i]);
        else
            AddGroupNarne(lenum.lana[i], szLocalName, &dwNum[i]);

// Отправка дейтаграмм
//
if (bSender)                  *
{
    // Отправка с использованием широковещания .ще-
    //
    if (bBroadcast)           ti
    {
        if (bOneLana)
        n <
            // Широковещательная отправка сообщения только на одном номере
            // LANA            и

```

. След. стр.

Листинг 1-5. (продолжение)

```

for(j = 0; j < dwNumDatagrams; j++)

    wsprintf(szMessage,
        "[J03d] Test broadcast datagram", j);
    if (DatagramSendBC(dwOneLana, dwNum[0],
        szMessage, strlen(szMessage))
        != NRC_GOODRET)
        return 1;
    Sleep(dwDelay);

}
else
{
    // Широковещательная отправка сообщения на каждом номере LAN^
    // на локальной машине
    for(j = 0; j < dwNumDatagrams; j++)
    {
        for(i = 0; i < lenum.length; i++)
        {
            wsprintf(szMessage,
                "[J03d] Test broadcast datagram", j);
            if (DatagramSendBC(lenum.lana[i], dwNum[i],
                szMessage, strlen(szMessage))
                != NRC_GOODRET)
                return 1;
            Sleep(dwDelay);
        }
    }
}
else
{
    // Отправка направленного сообщения на один узел
    for(j = 0; j < dwNumDatagrams; j++)
    {
        wsprintf(szMessage,
            "[X03d] Test directed datagram", j);
        if (DatagramSend(dwOneLana, dwNum[0], (jp,
            szRecipientName, szMessage,
            strlen(szMessage)) != NRC_GOODRET)
            return 1;
        Sleep(dwDelay);
    }
}
else
    rAil f*il<<J: 1<)\n*

```

## Листинг 1 -5. (продолжение)

```

// Отправка прямого сообщения на каждом номере LANA на
// локальной машине

for(j = 0; j < dwNumDatagrams;
{
    for(i = 0; i < lenum.length;

        wsprintf(szMessage,
            "[K03d] Test directed datagram", j);
        printf("count: Xd.JSd\n", j,i);
        if (DatagramSend(lenum.lana[i], dwNum[i],
            szRecipientName, szMessage,
            strlen(szMessage)) != NRCJ300DRET)
            return 1;

    }
    Sleep(dwDelay);
}
}
1 >

else // Прием дейтаграмм
{
    NCB *ncb=NULL;
    char <<szMessageArray = NULL;
    HANDLE *hEvent=NULL;
    DWORD dwRet;

    // Выделение массива структур NCB для передачи каждому приемнику
    // на каждом LANA
    //
    neb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
    //
    // Выделение массива буферов входящих данных
    //
    szMessageArray = (char **)GlobalAlloc(GMEM_FIXED,
        sizeof(char *) * lenum.length);
    for(i = 0; i < lenum.length; i++)
        szMessageArray[i] = (char *)GlobalAlloc(GMEM_FIXED,
            MAX_DATAGRAM_SIZE);
    //
    // Выделение массива описателей событий для
    // асинхронных приемов
    //
    hEvent = (HANDLE *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(HANDLE) * lenum.length);
    for(i = 0; i < lenum.length; i++)

```

} см. след. стр.



## Листинг 1-5. (продолжение)

```

        hEvent[i] = CreateEvent(0, TRUE, FALSE, 0);

if (bBroadcast)
{
    if (bOneLana)
    {
        // Синхронный широковещательный прием
        // на одном указанном номере LANA

        for(j = 0; j < dwNumDatagrams;

            if (DatagramRecvBC(&ncb[0], dwOneLana, dwNum[0],
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)                '***
                return 1;
            FormatNetbiosName(ncb[0].ncb_callname, szSender);
            " printf("X03d [LANA Xd] Message: 'Xs' "
                "received from: Xs\n", j,
                * ncb[0].ncb_lana_num, szMessageArray[0],      {
                szSender);                                         {

        >
    > else
    {
        // Асинхронный широковещательный прием на каждом
        // доступном номере LANA. Для каждой успешной команды
        // печатается сообщение, иначе команда отменяется.

        for(i = 0; j < dwNumDatagrams;

            for(i = 0; i < lenum.length; i++)                \\

                dwBytesRead = MAX_DATAGRAM_SIZE;
                if (DatagramRecvBC(&ncb[i], lenum.lana[i],    \\
                    dwNumfil, szMessageArray[i],              ,кмцл \\
                    MAX_DATAGRAM_SIZE, hEvent[i])              " *$
                    != NRC_GOODRET)
                    return 1;

                dwRet = WaitForMultipleObjects(lenum.length,
                    hEvent, FALSE, INFINITE);
                if (dwRet == WAIT_FAILED)                      \\

                <
                pnntfC'WaitForMultipleObjects failed: Xd\n", t \\
                    GetLastError());                            \\
                return 1;                                       K/v3rf

        >
        for(i = 0; i < lenum.length; i++) *, ,u ** I ,0

```

## Листинг 1-5. (продолжение)

```

        if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
            Cancel(&ncb[i]);
        else
        {
            ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
            FormatNetbiosName(ncb[i].ncb_callname,
                szSender);
            printf("X03d [LANA Xd] Message: 'Xs' "
                "received from: Xs\n", j,
                ncb[i].ncb_lana_num,
                szMessageArray[i], szSender);

            ResetEvent(hEvent[i]);
        }
    }
}
else
{
    if (bOneLana)
    {
        // Блокирующий прием дейтаграмм на указанном
        // номере LANA
        //
        for(j = 0; j < dwNumDatagrams; j++)
            if (bRecvAny)

                // Прием данных, предназначенных для любого имени NetBIOS
                // в таблице имен этого процесса

                if (DatagramRecv(&ncb[0], dwOneLana, 0xFF,
                    szMessageArray[0], MAX_DATAGRAM_SIZE,
                    NULL) != NRC_GOODRET)
                        return 1;
    }
    else
    {
        if (DatagramRecv(&ncb[0], dwOneLana,
            dwNum[0], szMessageArray[0],
            MAX_DATAGRAM_SIZE, NULL)
            != NRC_GOODRET)
                return 1;
    }
    FormatNetbiosName(ncb[0].ncb_callname, szSender);
    printf("X03d [LANA Xd] Message: 'Xs' "
        "received from: Xs\n", j,
        ncb[0].ncb_lana_num, szMessageArray[0],

```

см. след. стр.

## Листинг 1-5. (продолжение)

```

        szSender);

else

    // Асинхронный прием дейтаграмм на каждом доступном номере LANA.
    // Для успешных команд печатаются данные,
    // иначе команда отменяется.

    for(j = 0; j < dwNumDatagrams; j)
    {
        for(i = 0; i < lenum.length;
        {
            If (bRecvAny)

                // Прием данных, предназначенных для любой
                // NetBIOS в таблице имен этого процесса {
                //                                     tele
                if (DatagramRecv(&ncb[i], lenum.lana[i], }
                    OxFF, szMessageArray[i], i
                    MAX_DATAGRAM_SIZE, hEvent[i]) } ^
                    != NRC_GOODRET)
                        return 1;

            { 'ft
            if (DatagramRecv(&ncb[i], lenum.lana[i],
                dwNum[i], szMessageArray[i],
                MAX_DATAGRAM_SIZE, hEvent[i])
                != NRC_GOODRET)
                    return 1;

            } ft>
        }
    }
    dwRet = WaitForMultipleObjects(lenum.length,
        hEvent, FALSE, INFINITE);
    if (dwRet == WAIT_FAILED)
    {
        printf("WaitForMultipleObjects failed: %d\n",
            GetLastErrorQ);
        return 1;
    }

    for(i = 0; i < lenum.length;

        if (ncb[i].ncb_cmd_cplt == NRC_PENDING) iV"-
            Cancel(&ncb[i]);
        else
        {
            ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
            FormatNetbiosName(ncb[i].ncb_callname,

```

**Листинг 1-5. (продолжение)**

```

        szSender);
    printf("X03d [LANA Xd] Message: 'Xs' "
        "from: Xs\n", j, nob[i].ncb_lana_num,
        szHessageArray[i], szSender);
    }
    ResetEvent(hEvent[i]);

// Очистка
//
for(i = 0; i < lenum.length;

    CloseHandle(hEvent[i]);
    GlobalFree(szMessageArray[i]);
}
-f    GlobalFree(hEvent);
f.    GlobalFree(szMessageArray);
п    >
    // Очистка
•т И
    if (bOneLana)
        DelName(dwOneLana, szLocalName);
x    else
31    for(i = 0; i < lenum.length;
^        DelName(lenum.lana[i], szLocalName);
    }
-t    GlobalFree(dwNum);

    return 0;
}

```

Скомпилировав этот пример, выполните следующие тесты, чтобы понять, как работают дейтаграммы (табл. 1-2). В целях обучения запустите две копии приложений на разных компьютерах. Если вы запустите их на одном компьютере, они будут работать, но вы можете не увидеть некоторые важные моменты, так как в этом случае LANA для обеих сторон будут соответствовать одному и тому же протоколу.

**Табл. 1-2. Команды вызова Nbdgram.c**

<b>Команда клиента</b>	<b>Команда сервера</b>
Nbdgram /n:CLIENTO I	Nbdgram /s /n:SERVER01 /r-CLIENTO I
Nbdgram /n:CLIENT01 /b	Nbdgram /s /n:SERVER01 /b
Nbdgram /g:CLIENTGROUP	Nbdgram /s /r.CLIENTGROUP
Nbdgram /g:CLIENTGROUP	Nbdgram /s /r.CLIENTGROUP

А теперь перечислим все параметры командной строки, доступные для использования в программе-примере:

**Ш /n:my-name** — регистрирует уникальное имя *my-name* \

**Ж /g:group-name** — регистрирует групповое имя *group-name*;

**Ш /s** — отправляет дейтаграммы (по умолчанию, в примере **дейтаграммы** принимаются);

- **/c:n** — отправляет или получает *n* дейтаграмм;
- **/r:receiver** — определяет NetBIOS-имя для отправки дейтаграммы;

**Ш /Б** — использует широковещательную посылку дейтаграмм;

**Ж /a** — принимает данные для любого имени NetBIOS (присваивает полю *ncbjum* значение 0xFF);

**Ш /l:n** — выполняет все операции только на номере LANA *n* (по умолчанию все команды отправки и приема выполняются для всех номеров LANA);

- **/d:n** — ждет *n* миллисекунд между отправками.

Для выполнения третьей команды запустите несколько клиентов на разных компьютерах. Она иллюстрирует случай, когда один сервер посылает одно сообщение группе, и каждый член группы, ожидающий данные, получит это сообщение.

Также попробуйте различные комбинации перечисленных команд с параметром командной строки **/I-x**, где *x* — действительный номер LANA. Этот параметр переключает программу из режима выполнения команд на всех номерах LANA в режим выполнения команд только на конкретном номере LANA. Например, команда **Nbdgram /n:CLIENT01 /l:0** заставит приложение слушать входящие дейтаграммы только на LANA 0 и игнорировать любые данные, поступающие на какой-либо другой номер LANA.

Параметр **/a** имеет смысл только для клиентов. Этот флаг заставляет команду получения принимать входящие дейтаграммы, предназначенные для любого имени NetBIOS, которое зарегистрировано процессом. В нашем примере это не имеет особого значения, так как клиенты регистрируют только одно имя, но вы можете по крайней мере посмотреть, как это должно выглядеть в коде. Попробуйте изменить код так, чтобы он регистрировал имя для каждого параметра **/n-.name** в командной строке. Запустите сервер с флагом получателя, заданным только для одного из имен, зарегистрированных клиентом. Клиент получит данные, несмотря на то, что команда **NCBDGRECV** не обращается явно к конкретному имени.

## Дополнительные команды NetBIOS

s r

Все рассмотренные нами команды так или иначе относятся к установлению сеанса, отправке или получению данных через сеанс или дейтаграмму. Но есть несколько команд, предназначенных исключительно для обработки приема информации: команда опроса состояния адаптера (**NCBASTAT**) и команда поиска имени (**NCBFJNDNAME**). Мы рассмотрим их, а затем перейдем

к программному сопоставлению номеров LANA протоколам — хотя это не функция NetBIOS, но в справочных целях ознакомиться с ней полезно.

## Проверка состояния адаптера (команда *NCBASTAT*)

Команда опроса состояния адаптера полезна для получения информации о локальном компьютере и его номерах LANA. Это единственный способ программно выяснить MAC-адрес компьютера из Windows 95 и NT 4. Функции IP Helper, появившиеся в Windows 2000 и Windows 98 (см. приложение А), предоставляют более универсальный интерфейс для выяснения MAC-адреса.

Команда и ее синтаксис довольно просты, но то, какие данные будут возвращены, зависит от способа вызова функции. Команда опроса состояния адаптера возвращает структуру *ADAPTER\_STATUS*, сопровождаемую рядом структур *NAME\_BUFFER*. Эти структуры определены следующим образом:

```
typedef struct _ADAPTER_STATUS {
    UCHAR    adapter_address[6];
    UCHAR    rev_major;
    UCHAR    reserved0;
    UCHAR    adapter_type;
    UCHAR    rev_minor;
    WORD     duration;
> « WORD    frmr_recv;
    WORD     frmr_xmit;
    WORD     iframe_recv_err;
    WORD     xmit_aborts;
    DWORD    xmit_success;
    DWORD    recv_success;
    WORD     iframe_xmit_err;
    WORD     recv_buff_unavail;
    WORD     t1_timeouts;
    WORD     ti_timeouts;
    DWORD    reserved1;
    WORD     free_ncbs;
    WORD     max_cfg_ncbs;
    WORD     max_ncbs;
    WORD     xmit_buf_unavail;
    WORD     max_dgram_size;
    WORD     pending_sess;
    WORD     max_cfg_sess;
    WORD     max_sess;
    WORD     max_sess_pkt_size;
    WORD     name_count;
> ADAPTER_STATUS, *PADAPTER_STATUS;

typedef struct _NAME_BUFFER {
    UCHAR    name[NCBNAHSZ];
    UCHAR    name_num;
    UCHAR    name_flags;
} NAME_BUFFER, *PNAME_BUFFER;
```

Поля, представляющие наибольший интерес: MAC-адрес (*adapter\_address*), максимальный размер дейтаграммы (*jmax\_dgram\_size*) и максимальное количество сеансов (*max\_sess*). Кроме того, поле *name\_count* сообщает, сколько было возвращено структур *NAME\_BUFFER*. Максимальное количество NetBIOS-имен на номер LANA — 254, так что вы можете обеспечить достаточно большой буфер для всех имен или вызвать команду опроса состояния адаптера с полем *ncbjlength*, равным 0. Функция *Netbios* по завершении выдает необходимый размер буфера.

Поля, необходимые для вызова команды *NCBASTAT*-. *ncb\_command*, *ncb\_buffer*, *ncbjlength*, *ncbjanum* и *ncbjzallname*. Если первый символ поля *ncb\_callname* — звездочка (\*), команда проверки состояния выполняется, но возвращает только NetBIOS-имена, добавленные вызывающим процессом. Впрочем, если вы вызываете *Netbios* с командой опроса состояния адаптера, добавляете уникальное имя к таблице имен текущего процесса, а затем используете это имя в поле *ncb\_callname*, все NetBIOS-имена, а также все имена, зарегистрированные системой, добавляются в таблицу имен локального процесса. Вы можете также проверить состояние адаптера не с того компьютера, где выполняется команда. Для этого задайте в поле *ncb\_callname* имя удаленной рабочей станции.

**ПРИМЕЧАНИЕ** Помните, что во всех именах компьютеров Microsoft 16-му байту присваивается значение 0. Кроме того, они дополняются пробелами до фиксированной длины.

Приведенная в качестве примера простая программа — *Astat.c*, проверяет состояние всех LANA. Кроме того, при использовании флага */l:LOCAL-NAME* эта команда выполняется на локальном компьютере, но выдает полную таблицу имен. Флаг */r:REMOTENAME* инициирует удаленный опрос для указанного имени компьютера.

При использовании команды, проверяющей состояние адаптера, нужно учитывать некоторые моменты. Во-первых, у компьютера с несколькими адаптерами будет несколько MAC-адресов. Так как NetBIOS не позволяет выяснить, к которым адаптерам и протоколам привязан LANA, разбираться в возвращенных значениях придется самостоятельно. Кроме того, если установлена *служба удаленного доступа* (Remote Access Service, RAS), система выделяет номера LANA и для удаленных соединений. Пока соединения RAS неактивны, проверка состояния адаптера на этих LANA вернет нулевой MAC-адрес. Если установлено соединение RAS, MAC-адрес будет соответствовать тому, который служба RAS назначает всем своим виртуальным сетевым устройствам.

Наконец, проверку состояния удаленного адаптера вы должны выполнять по общему для обоих компьютеров транспортному протоколу. Например, системная команда *Nbtstat* (версия *NCBASTAT* для командной строки) выполняет опрос только по TCP/IP. Если на удаленном компьютере нет TCP/IP, команда не будет выполнена.

## Команда поиска имени {NCBFINDNAME}

Эта команда доступна только в Windows NT и Windows 2000, сообщает, кто зарегистрировал данное имя NetBIOS. Чтобы успешно выполнить запрос на поиск имени, процесс должен добавить свое уникальное имя в таблицу имен. Для этого необходимо задать следующие поля: команда, номер LANA, буфер и длина буфера. Запрос вернет структуру *FIND\_NAME\_HEADER* и несколько структур *FIND\_NAME\_BUFFER*, определенных следующим образом:

```
typedef struct _FIND_NAME_HEADER {
    WORD    node_count;
    UCHAR   reserved;
    UCHAR   unique_group;
} FIND_NAME_HEADER, *PFIND_NAME_HEADER;

typedef struct _FIND_NAME_BUFFER {
    UCHAR   length;
    UCHAR   access_control;
    UCHAR   frame_control;
    UCHAR   destination_addr[6];
    UCHAR source_addr[6];
    UCHAR   routing_info[18];
} FIND_NAME_BUFFER, *PFIND_NAME_BUFFER;
```

Как и в случае команды проверки состояния адаптера, если команда *NCB-FINDNAME* выполняется с длиной буфера равной 0, функция *Netbios* вернет требуемую длину и ошибку *NRC\_BUFLLEN*.

Структура *FIND\_NAME\_HEADER*, которую возвращает успешный опрос, показывает, зарегистрировано ли имя как уникальное или как групповое. Если поле *unique\_group* содержит 0 — это уникальное имя, если 1 — групповое. Поле *node\_count* указывает, сколько было возвращено структур *FIND\_NAME\_BUFFERS*. Структура *FIND\_NAME\_BUFFER* возвращает совсем немного информации, большая часть которой полезна на уровне протокола. Однако нас интересуют поля *destination\_addr* и *source\_addr*. Поле *source\_addr* содержит MAC-адрес сетевого адаптера, зарегистрировавшего имя, а поле *destination\_addr* — MAC-адрес адаптера, выполнившего запрос.

Запрос на поиск имени может быть отдан на любом номере LANA на локальном компьютере. Возвращенные данные должны быть одинаковы на всех действительных LANA в локальной сети (вы можете выполнить команду поиска имени для RAS-подключения, чтобы определить, зарегистрировано ли имя в удаленной сети).

Если в Windows NT 4 поиск имени выполняется по протоколу TCP/IP, *Netbios* возвращает ложную информацию. Убедитесь, что выбран номер LANA, соответствующий транспорту, отличному от TCP/IP.

## Сопоставление протоколов номерам LANA

В зависимости от того, какой транспорт использует ваше приложение, могут возникать разные проблемы, так что неплохо уметь программно опре-



делять доступные транспорты. Это невозможно сделать средствами «родного» опроса NetBIOS — надо использовать Winsock 2 для Windows NT 4 и Windows 2000. Функция Winsock 2 *WSAEnumProtocols* возвращает информацию о доступных транспортных протоколах (подробнее о ней — в главах 5 и 6). Хотя Winsock 2 доступен в Windows 95 и, по умолчанию, в Windows 98, информация о протоколе, хранящаяся на этих платформах, не содержит никаких сведений о NetBIOS.

Мы не будем подробно обсуждать Winsock 2, поскольку этому интерфейсу посвящена часть II этой книги. Основные шаги следующие: загрузка Winsock 2 через функцию *WSAStartup*, вызов функции *WSAEnumProtocols* и просмотр возвращенных запросом структур *WSAPROTOCOLINFO*. Пример NbpProto.c на прилагаемом компакт-диске содержит код для выполнения такого опроса.

Функция *WSAEnumProtocols* принимает в качестве параметров адрес буфера для блока данных и длину буфера. Сначала вызовите эту функцию с нулевыми адресом и длиной буфера. Вызов будет неудачным, но параметр длины буфера теперь содержит требуемый размер. Вызовите функцию снова — *WSAEnumProtocols* вернет количество найденных протоколов. Структура *WSAPROTOCOLINFO* велика и содержит множество полей, но нас интересуют только *szProtocol*, *iAddressFamily* и *iProtocol*. Если поле *iAddressFamily* равно *AF\_NETBIOS*, то абсолютное значение *iProtocol* — номер LANA для протокола, указанного в строке *szProtocol*. Кроме того, для сопоставления возвращенного протокола predeterminedенному GUID протокола можно использовать *ProviderId* GUID.

Здесь есть один нюанс. В Windows NT и Windows 2000 поле *iProtocol* для любого протокола, установленного на LANA 0, имеет значение 0x80000000. Дело в том, что протокол 0 зарезервирован для специального использования; любой протокол, назначенный LANA 0, всегда будет иметь значение 0x80000000, так что надо просто проверить это значение.

## Рекомендации по выбору платформ

Реализуя связь средствами NetBIOS на следующих платформах, имейте в виду следующие ограничения.

### Платформа Windows CE

Интерфейс NetBIOS в Windows CE не доступен. Перенаправитель поддерживает имена и разрешения имен NetBIOS, но не программный интерфейс.

### Платформа Windows 9x

В Windows 95 и Windows 98 есть несколько ошибок. На любой из этих платформ вы должны сбросить все номера LANA перед добавлением любого имени NetBIOS к любому номеру LANA. Так как сброс одного номера LANA разрушает таблицы имен других номеров, избегайте кода, подобного следующему.

```
iANA_ENUM    lenum;  
// Перечисление номеров LANA  
for(i = 0; i < lenum.length;  
<  
    Reset(lenum.lana[i]);  
    AddName(lenum.lana[i], MY_NETBIOS_NAME);  
}
```

Кроме того, не пытайтесь асинхронно выполнить в Windows 95 команду *NCBRESET* на соответствующем протоколу TCP/IP номере LANA. Для начала, не следует отдавать эту команду асинхронно, так как прежде, чем вы сможете сделать что-нибудь с этим номером LANA, должен завершиться сброс. При асинхронном выполнении команды *NCBRESET* приложение вызовет фатальную ошибку в драйвере виртуального устройства (virtual device driver, VXD) NetBIOS TCP/IP и придется перезагружать компьютер.

## Для любых платформ

При сеансовой связи одна сторона может посылать сколь угодно много данных, однако отправитель на деле буферизует посылаемые данные, пока получатель не подтвердит их получение, отдав команду приема. NetBIOS-команды *NCBSENDNA* и *NCBCHAINSENDNA* — варианты команд отправки, не требующие подтверждения. Вы можете использовать их, если намеренно не хотите, чтобы команда отправки ждала подтверждения от получателя. Поскольку в нижележащем протоколе TCP/IP реализована собственная схема подтверждения, команды отправки, не требующие подтверждения от получателя, ведут себя так же, как и ожидающие подтверждения.

## Резюме

NetBIOS — мощный, но устаревший прикладной интерфейс. Одна из его сильных сторон — независимость от протокола: приложения могут работать поверх TCP/IP, NetBEUI и SPX/IPX. NetBIOS обеспечивает связь с установлением логического соединения и без такового. Значительное преимущество интерфейса NetBIOS перед Winsock — единый способ разрешения имен и регистрации. Приложение NetBIOS нуждается только в имени NetBIOS, а приложение Winsock, использующее разные протоколы, должно знать схему адресации каждого (см. часть II).

В главе 2 речь пойдет о перенаправителе — составной части почтовых ящиков и именованных каналов (о них — в главах 3 и 4).

## Перенаправитель

Microsoft Windows позволяет приложениям обмениваться информацией по сети с помощью встроенных служб файловой системы, иногда называемых *сетевой операционной системой* (network operating system, NOS). В этой главе описываются сетевые возможности, использующие компоненты файловой системы Windows, и доступные в Windows 95, Windows 98, Windows NT, Windows 2000 и Windows CE. Они основаны на сетевых технологиях *почтовых ящиков* (mailslot) и *именованных каналов* (named pipe), о которых речь пойдет в главах 3 и 4.

Для доступа к локальным файлам приложения посылают запросы ввода-вывода операционной системы (обычно это называется локальным вводом-выводом). Например, когда приложение открывает или закрывает файл, ОС определяет устройство, на котором находится данный файл, и передает запрос ввода-вывода локальному драйверу этого устройства. Аналогично осуществляется доступ к устройствам по сети, только запрос ввода-вывода передается по сети удаленному устройству. Это называется *перенаправлением ввода-вывода* (I/O redirection). Например, Windows позволяет назначить имя локального диска (скажем, E:) общей папке на удаленном компьютере. Тогда при обращении приложений к диску E: ОС будет перенаправлять запросы ввода-вывода на устройство, называемое *перенаправителем* (redirector), а он — формировать канал связи к удаленному компьютеру для доступа к нужной общей папке. Это позволяет приложениям использовать для доступа к файлам по сети обычные API-функции для работы с файлами (типа *ReadFile* и *WriteFile*).

В этой главе подробно рассматривается использование перенаправителя для передачи запросов ввода-вывода на удаленные устройства (именно на этом основана связь в технологиях почтовых ящиков и именovaných каналов). Сначала мы обсудим, как ссылаться на файлы в сети с помощью *универсальных правил именования* (Universal Naming Convention, UNC) и указателя ресурса *поставщика нескольких UNC* (Multiple UNC Provider, MUP). Затем поясним, как MUP вызывает сетевую службу, которая использует перенаправитель для установления связи между компьютерами по протоколу Server Message Block (SMB). В заключение — вопросы обеспечения безопасности при доступе к файлам по сети с помощью базовых операций файлового ввода-вывода.

## Универсальные правила именования

Имена UNC — это стандартный способ доступа к файлам и устройствам без назначения этим объектам буквы локального диска, спроецированного на удаленную файловую систему. Это позволяет приложениям не зависеть от имен дисков и прозрачно работать с сетью. В частности, не надо беспокоиться, что не хватит букв для подключаемых общих ресурсов. К тому же структура подключенных дисков своя у каждого пользователя, и процессы, запущенные в другом контексте, не смогут обратиться к вашим сетевым дискам.

Имена UNC имеют вид

```
\\сервер\ресурс\путь
```

Первая часть — *\\сервер*, начинается с двух обратных косых черт и имени удаленного сервера, на котором находится нужный файл. Вторая — *ресурс*, это имя общего ресурса, то есть папки в файловой системе, к которой открыт общий доступ пользователям сети. Третья часть — *путь*, обозначает путь к нужному файлу. Предположим, на сервере с именем Myserver находится папка D:\Myfiles\CoolMusic, предоставленная для общего доступа под сетевым именем Myshare, а в этой папке — файл Sample.mp3. Тогда для доступа к этому файлу с другой машины надо указать следующее UNC-имя.

```
\\Myserver\Myshare\Sample.mp3
```

Как видите, это способ гораздо проще, чем подключение общей папки Myshare в качестве сетевого диска.

Обращение к файлам по сети с помощью UNC-имен скрывает от приложения детали формирования сетевого соединения, так что система легко находит нужные файлы даже при подключении по модему. Все детали сетевого соединения организуются перенаправителем компонента доступа.

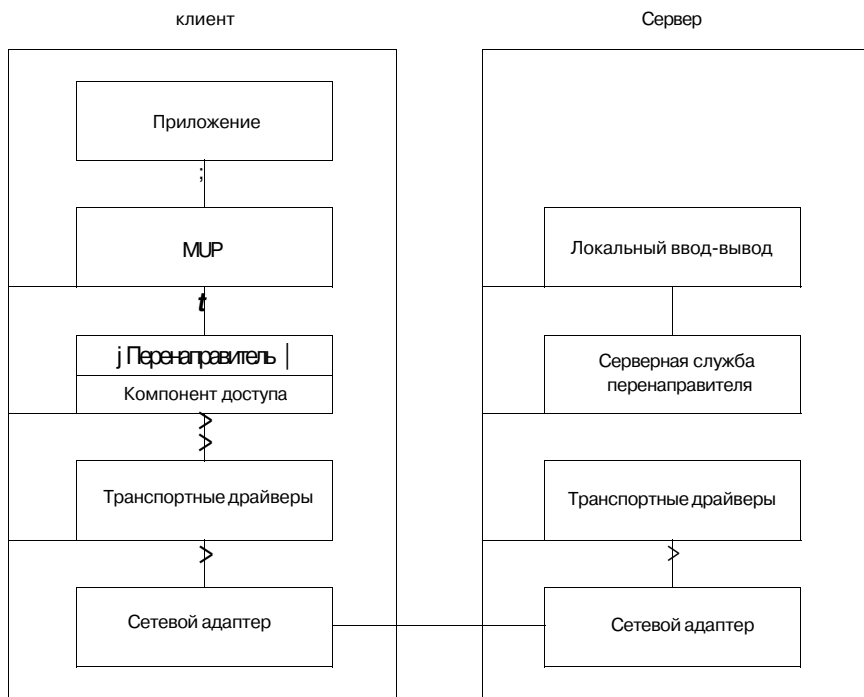
На рис. 2-1 изображены основные компоненты, формирующие UNC-соединение в NOS в рамках Windows, а также показано, как перемещаются данные между компонентами клиента и сервера NOS.

## Поставщик нескольких UNC

MUP — указатель ресурса, ответственный за выбор компонента доступа для обслуживания соединений по UNC-именам. Компонент сетевого доступа, или *сетевой поставщик* (network provider) — это служба, способная использовать сетевые устройства для доступа к ресурсам, расположенным на удаленном компьютере, например, к файлам и принтерам. MUP использует компонент сетевого доступа для организации связи в ответ на все запросы ввода-вывода к файлам и принтерам по UNC-именам.

В Windows NT, Windows 2000, Windows 95 и Windows 98 может быть несколько компонентов доступа одновременно. Так, в платформы Windows встроен клиент для сетей Microsoft (Client for Microsoft Networks), но можно также установить сторонние компоненты доступа, например, клиент для сетей Novell (Novell Client v3.01 for Windows 95/98). Таким образом, обслужить UNC-запрос могут несколько компонентов одновременно. С другой

стороны, в Windows CE имеется только один поставщик — клиент для сетей Microsoft.



**Рис. 2-1. Компоненты перенаправителя**

Главная функция MUP — выбор сетевого компонента, который должен обслужить UNC-запрос. MUP делает это, параллельно посылая UNC-имена из запроса всем установленным компонентам доступа. Если какой-либо компонент отвечает, что способен обслужить запрос с данными именами, то ему направляется весь запрос. Если это могут сделать несколько компонентов, то MUP выбирает тот, у которого наивысший приоритет. Приоритет компонентов определяется порядком, в котором они были установлены в системе. В Windows NT, Windows 2000, Windows 95 и Windows 98 этот приоритет определяет параметр `ProviderOrder` в реестре Windows в разделе `\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order`.

Установленные компоненты перечислены в порядке приоритета. Поскольку в Windows CE только один компонент доступа, MUP не используется, и UNC-запросы передаются сразу этому компоненту.

## Компоненты сетевого доступа

Напомним, что компонент сетевого доступа — это служба, использующая сетевые устройства для доступа к файлам и принтерам на удаленном компьютере, сердцевина NOS. Компонент доступа может, например, перенаправить идентификатор локального диска (например, E:) на общую папку на удаленном компьютере. Компоненты доступа должны также обслуживать запросы

на соединение по UNC-именам. В Windows они используют для этого перенаправитель.

В комплект Windows входит клиент для сетей Microsoft (Client for Microsoft Networks), ранее называвшийся Microsoft Networking Provider (MSNP), который обеспечивает связь между Windows NT 4, Windows 2000, Windows 95, Windows 98 и Windows CE. Последняя, однако, не поддерживает установку нескольких клиентов сети одновременно и имеет встроенную поддержку лишь для клиента MSNP.

## Перенаправитель

Этот компонент ОС использует для приема и обработки запросов ввода-вывода: перенаправитель формирует запросы и отправляет их серверной службе на удаленном компьютере, которая генерирует локальные запросы ввода-вывода. Поскольку перенаправитель предоставляет средства ввода-вывода службам верхнего уровня (типа MUP), он скрывает детали сетевого уровня от приложений. Поэтому приложения не должны предавать перенаправителю параметры, связанные с протоколами. В итоге компонент сетевого доступа не зависит от протокола: приложения могут работать практически в любой сетевой конфигурации.

MSNP содержит перенаправитель, напрямую взаимодействующий с уровнем сетевого транспорта, и NetBIOS для обеспечения связи между клиентом и сервером. NetBIOS API (см. главу 1), предоставляет интерфейс программирования этих транспортов. Перенаправитель MSNP часто называют перенаправителем диспетчера ЛВС (LAN Manager), поскольку он разрабатывался на основе старого диспетчера ЛВС Microsoft, обеспечивавшего работу в сети приложениям MS-DOS. Интерфейс NetBIOS позволяет установить связь по самым разным протоколам, что делает перенаправитель MSNP протоколоне зависым. Приложениям, которые его используют, не нужно знать детали работы сетевых протоколов: они выполняются поверх TCP/IP, NetBEUI или даже IPX/SPX. Это замечательная возможность позволяет приложениям обмениваться информацией, независимо от физической организации сети. Впрочем, есть один нюанс: чтобы два приложения могли связаться по сети, их рабочие станции должны иметь хотя бы один общий протокол. Так, если на компьютере А установлен только TCP/IP, а на компьютере В — только IPX, перенаправитель MSNP не сможет обеспечить связь между ними.

Перенаправитель MSNP связывается с другими рабочими станциями, посылая сообщения серверной службе перенаправителя на этих станциях. Эти сообщения задают строго в определенной структуре, называемой Server Message Block (SMB). Протокол, по которому перенаправитель фактически отправляет и получает сообщения с удаленных компьютеров, называется Server Message Block File Sharing Protocol (протокол совместного использования файлов на основе блоков сообщений сервера) или просто SMB.

## Протокол SMB

Этот протокол был разработан Microsoft и Intel в конце 80-х гг. для упрощения доступа приложений MS-DOS к удаленным файловым системам. Теперь

он просто позволяет перенаправителю MSNP в Windows использовать структуру данных SMB при связи со службой сервера MSNP на удаленной рабочей станции. Структура данных SMB содержит три основных компонента: код команды, параметры команды и пользовательские данные.

Протокол SMB основан на простой модели запросов клиента и ответов сервера. Клиент перенаправителя MSNP создает структуру SMB с указанием типа запроса в поле кода команды. Если команда требует отправки данных (например, SMB-команда Write), то они прилагаются к запросу. Затем структура SMB отправляется по транспортному протоколу (например, TCP/IP) серверной службе на удаленной рабочей станции. Эта служба обрабатывает запрос клиента и возвращает ему ответную структуру SMB.

Теперь рассмотрим пример: открытие файла \\Myserver\Myshare\Sample.mp3 по сети. При этом происходит следующее.

1. Приложение направляет запрос на открытие этого файла локальной ОС с помощью API-функции *CreateFile*.
2. Локальная ОС определяет по UNC-имени, что этот запрос ввода-вывода адресован удаленной машине с именем \\Myserver, и передает его MUP.
3. MUP определяет, что запрос предназначен компоненту доступа MSNP, поскольку именно этот компонент обнаруживает \\Myserver путем разрешения NetBIOS-имени.
4. Запрос ввода-вывода передается перенаправителю MSNP.
5. Перенаправитель форматирует запрос на открытие файла Sample.mp3 в удаленной папке \Myshare как сообщение SMB.
6. Форматированное сообщение SMB передается по сетевому транспортному протоколу.
7. Сервер с именем \\Myserver получает SMB-запрос по сети и передает его серверной службе своего перенаправителя MSNP.
8. Серверная служба выполняет локальный запрос ввода-вывода на открытие файла Sample.mp3 в общей папке \Myshare.
- 9- Перенаправитель сервера форматирует ответное SMB-сообщение с информацией об успехе или неудаче локального запроса ввода-вывода на открытие файла.
10. Ответ сервера посылается по сетевому транспортному протоколу обратно клиенту.
11. Перенаправитель MSNP получает ответ SMB и передает код возврата локальной ОС.
12. Локальная ОС передает код возврата вызвавшему функцию *CreateFile* приложению.

Как видите, перенаправителю MSNP требуется всего несколько шагов, чтобы предоставить приложению доступ к удаленным ресурсам. Перенаправитель также управляет доступом к ресурсам, что является одной из форм сетевой безопасности.

# Безопасность

Прежде чем приступить к теме безопасности и правил доступа к ресурсам в сети, обсудим основы обеспечения безопасности на локальной машине. Windows NT и Windows 2000 позволяют управлять доступом к отдельным файлам и папкам как локально, так и по сети. При попытке приложения получить доступ к таким ресурсам ОС проверяет, имеет ли это приложение соответствующие права. Основные виды доступа — чтение, запись и выполнение. Windows NT и Windows 2000 управляют доступом на основе *дескрипторов безопасности* (security descriptors) и *маркеров доступа* (access tokens).

## Дескрипторы безопасности

Все защищенные объекты обладают дескриптором безопасности, содержащим информацию о порядке доступа к объекту. Дескриптор безопасности состоит из структуры *SECURITY\_DESCRIPTOR* и связанной с ним информации о безопасности, включающей:

**III идентификатор безопасности (Security Identifier, SID) владельца** — определяет владельца объекта;

- **SID группы** — определяет основную группу, в которую входит владелец объекта;

**III избирательный список управления доступом (Discretionary Access Control List, DACL)** — указывает, кто и какой тип доступа (чтение, запись, выполнение) имеет для данного объекта;

**III системный список управления доступом (system access control list, SACL)** — задает типы доступа к данному объекту, для которых генерируются записи в журнал аудита.

Приложения не могут напрямую изменять содержимое структуры дескриптора безопасности. Впрочем, для этого можно использовать API-интерфейсы безопасности Win32, содержащие соответствующие функции (мы продемонстрируем, как это сделать в конце главы).

## Списки и записи управления доступом

Поля DACL и SACL в дескрипторе безопасности — это *списки управления доступом* (access control lists, ACL), содержащие ноль или более *записей управления доступом* (access control entities, ACE). Каждая ACE управляет доступом или осуществляет контроль за доступом к объекту определенного пользователя или группы и содержит:

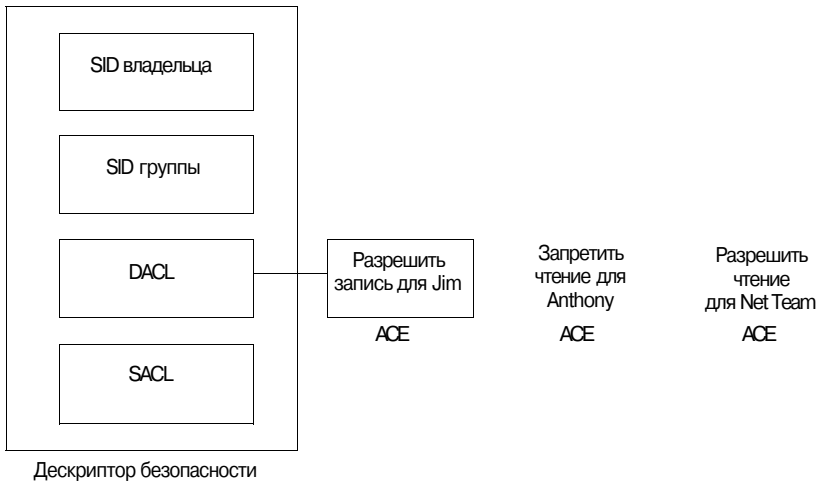
И SID пользователя или группы, к которым применяется ACE;

И маску, задающую права доступа (чтение, запись, выполнение);

- \* флаг, обозначающий тип ACE — разрешение на доступ, запрет на доступ или системный аудит.

Заметим, что системный аудит применяется только в списках SACL, а типы разрешение и запрет на доступ — в списках DACL (рис. 2-2).





**Рис. 2-2. Объект файла с соответствующим ему DACL**

Если защищенный объект не имеет списка DACL (его DACL был обнулен API-функцией *SetSecurityDescriptorDad*), то система предоставляет всем полный доступ. Если объект имеет непустой DACL, система предоставляет только те типы доступа, которые явно указаны в записях ACE данного DACL. Если в списке нет ни одной записи ACE, система не предоставляет никому никакого доступа. Если же DACL содержит несколько разрешающих доступ записей ACE, система неявно запрещает доступ всем пользователям и группам, не включенным в список.

В большинстве случаев задают только разрешающие доступ записи ACE, за одним исключением: если имеется разрешающая запись для группы, но нужно запретить доступ каким-либо членам этой группы с помощью ACE. При этом запрещающая доступ запись ACE должна обязательно предшествовать записи, открывающей доступ всей группе, поскольку система читает записи по порядку и прекращает чтение, выяснив, что доступ данному пользователю открыт или закрыт. То есть если запись, разрешающая доступ группе, будет идти первой, система прочтет ее и предоставит доступ неполномочному пользователю из состава группы.

На рис. 2-2 показан список DACL, предоставляющий доступ для чтения группе Net Team. Эта группа состоит из пользователей Anthony, Jim и Gary, и нужно предоставить право чтения всем, кроме Anthony. Поэтому запись, запрещающая доступ пользователю Anthony, предшествует записи, разрешающей доступ группе Net Team. На рис. 2-2 также показана запись ACE, предоставляющая пользователю Jim доступ для записи. Напомним, что приложения не могут напрямую работать с ACE, а используют для этого специальные API-интерфейсы.

## Идентификаторы безопасности

Дескрипторы безопасности и записи ACE защищенных объектов содержат SID — уникальный код для идентификации учетной записи пользователя,

группы или сеанса. Центр безопасности (например, домен Windows NT) хранит информацию о SID в специальной базе данных. Когда пользователь входит в систему, его SID извлекается из БД, помещается в маркер доступа пользователя и применяется для идентификации пользователя при всех проверках безопасности.

## Маркеры доступа

При входе пользователя в систему Windows NT проверяются его реквизиты: имя учетной записи и пароль. Если вход разрешен, система создает маркер доступа и заносит в него SID пользователя. Каждый процесс, запущенный от имени этого пользователя, получит копию маркера. При попытке доступа процесса к защищенному объекту SID в маркере доступа сравнивается с SID в списках DACL для определения прав доступа.

## Сетевая безопасность

Рассмотрим теперь обеспечение безопасности при доступе к объектам по сети. Напомним, что за доступ к ресурсам на удаленных компьютерах отвечает перенаправитель MSNP. Для этого он устанавливает безопасное соединение клиент-сервер, создавая реквизиты сеанса пользователя.

### Реквизиты сеанса

Существуют реквизиты пользователя двух типов: *основные реквизиты входа* (primary login) и реквизиты сеанса. Когда пользователь регистрируется на рабочей станции, вводимые им имя и пароль становятся его основными реквизитами входа и заносятся в маркер доступа. В один момент времени пользователь может иметь единственный набор реквизитов. При попытке доступа к удаленному ресурсу (как через сетевой диск, так и по UNC-имени) пользовательские реквизиты входа используются для проверки прав доступа к этому объекту.

В Windows NT и Windows 2000 можно указать другой набор реквизитов для удаленного доступа по сети. Если реквизиты пользователя действительны, перенаправитель MSNP создает сеанс связи между компьютером пользователя и удаленным ресурсом. При этом он сопоставляет сеансу реквизиты, состоящие из копии реквизитов, примененных компьютером пользователя для доступа к ресурсу. При каждом сеансе связи между компьютером и удаленным сервером используется только один набор реквизитов. Например, если на машине В имеются общие папки \Hack и \Slash, и пользователь с машины А подключает папку \Hack как диск G:, а папку \Slash — как диск H:, то оба сеанса связи будут применять одинаковые реквизиты сеанса, так как устанавливают соединение с одним и тем же удаленным сервером.

На удаленном сервере безопасность контролирует серверная служба перенаправителя MSNP. При попытке получить доступ к защищенному объекту, она использует реквизиты сеанса для создания маркера удаленного доступа. Далее безопасность обеспечивается так же, как и при локальном доступе (рис. 2-3)-

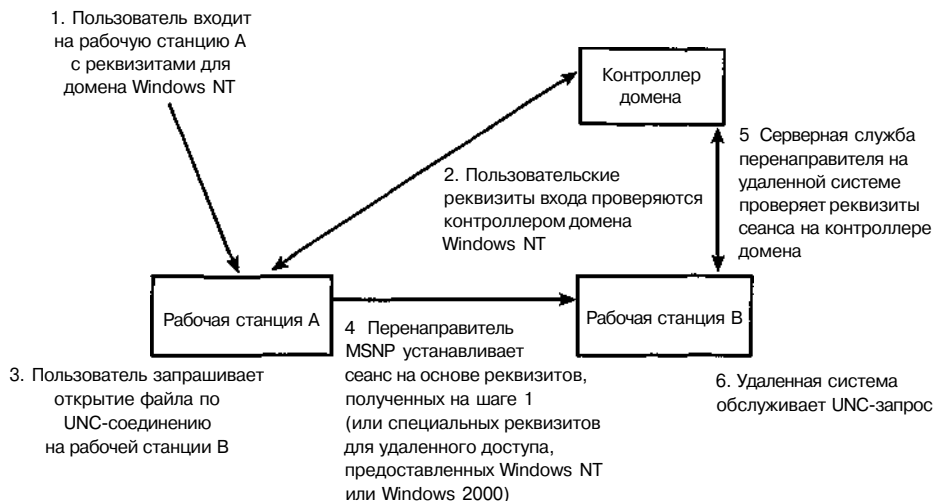


Рис. 2-3. Применение реквизитов безопасности

## Пример

Приложения Win32 могут использовать API-функции *CreateFile*, *ReadFile* и *WriteFile* для создания, открытия и изменения файлов по сети с использованием перенаправителя MSNP. Заметьте, что только платформы Windows NT и Windows 2000 поддерживают модель безопасности Win32. В листинге 2-1 приведен пример кода приложения, создающего файл по UNC-соединению. Вы можете найти его в папке \Examples\Chapter02 на прилагаемом компакт-диске.

Листинг 2-1. Простой пример создания файла

```

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

void main(void)

HANDLE FileHandle;
DWORD BytesWritten;

// Открытие файла \\Myserver\Myshare\Sample.txt
if ((FileHandle = CreateFile("\\\\Myserver\\Myshare\\Sample.txt",
    GENERIC_WRITE | GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL))
    == INVALID_HANDLE_VALUE)

    printf("CreateFile failed with error %d\n", GetLastError());
    return;
  
```

**Листинг 2-1. (продолжение)**

*// Запись 14 байт в новый файл*

```
if (WriteFile(FileHandle, "This is a test", 14,
    &BytesWritten, NULL) == 0)
{
    printf("WriteFile failed with error Xd\n", GetLastError());
    return;

if (CloseHandle(FileHandle) == 0)
{
    printf("CloseHandle failed with error Xd\n", GetLastError());
    return;
```

## Резюме

В этой главе вы познакомились с перенаправителем Windows, позволяющим приложениям получать доступ к ресурсам файловой системы Windows по сети. Мы рассказали, как он осуществляет обмен информацией по сети и как при этом используется система безопасности Windows NT и Windows 2000.

Две следующие главы посвящены технологиям почтовых ящиков и именovaných каналов, опирающихся на перенаправитель для выполнения любых действий по сети.

**At**

## Почтовые ящики

В Microsoft Windows NT, Windows 2000, Windows 95 и Windows 98 (но не Windows CE) реализован простой однонаправленный механизм *межпроцессной связи* (interprocess communication, IPC), называемый *почтовыми ящиками* (mailslots). Почтовые ящики позволяют клиентскому процессу передавать сообщения одному или нескольким серверным процессам. Они также помогают передавать сообщения между процессами на одном и том же компьютере или на разных компьютерах в сети. Разработка приложений, использующих почтовые ящики, не требует знания сетевых транспортных протоколов, таких как TCP/IP или IPX. Поскольку почтовые ящики основаны на архитектуре широковещания, они не гарантируют надежной передачи данных, но полезны, когда доставка данных не является жизненно важной.

Один из таких случаев — создание системы сообщений, охватывающей всех сотрудников офиса. Допустим, в офисе с множеством рабочих станций не хватает газированной воды, и все пользователи хотят каждые несколько минут знать, сколько банок кока-колы осталось в автомате. Легко установить клиентское приложение, отслеживающее количество банок и передающее эту информацию всем заинтересованным пользователям каждые пять минут. Поскольку почтовые ящики не гарантируют доставку широковещательного сообщения, некоторые пользователи могут не получить всех обновлений информации. Но несколько неудачных передач не приведут к неприятностям, потому что информация обновляется достаточно часто.

Итак, основное ограничение почтовых ящиков: они допускают только ненадежную однонаправленную передачу данных от клиента к серверу. Основное преимущество: клиентские приложения могут легко посылать широковещательные сообщения одному или нескольким серверным приложениям.

В этой главе мы расскажем о создании клиент-серверного приложения, правилах именования почтовых ящиков, влиянии размера сообщений на работу почтовых ящиков, связанных с ними проблемах и ограничениях.

## Подробности внедрения почтовых ящиков

Почтовые ящики основаны на интерфейсе файловой системы Windows. Клиентское и серверное приложения используют стандартные функции ввода-вывода файловой системы Win32 (такие как *ReadFile* и *WriteFile*) для отправки и получения данных почтовым ящиком, а также правила именования фай-

ловой системы Win32. Для создания и идентификации почтовых ящиков перенаправитель Windows использует файловую систему Mailslot File System (MSFS). Подробнее о перенаправителе Windows — в главе 2.

## Имена почтовых ящиков

Почтовые ящики именуются по следующему правилу:

```
\\сервер\Mailslot\[путь]имя
```

Строка состоит из трех частей: `\\server`, `\Mailslot` и `\[path]name`, где `\\сервер` — имя сервера, на котором создается почтовый ящик и выполняется серверное приложение; `\Mailslot` — фиксированная обязательная строка, уведомляющая систему, что это имя файла относится к MSFS; `\путь\имя` — позволяет приложениям уникальным образом определять и идентифицировать имя почтового ящика. В строке *путь* разрешается задавать несколько уровней каталогов. Например, допустимы следующие типы имен почтовых ящиков:

```
\\Oreo\Mailslot\Mymailslot
```

```
\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot
```

```
\\.\Mailslot\Easymailslot
```

```
\\*\Mailslot\Myslot
```

Строка *сервер* может представлять собой точку (.), звездочку (\*), имя домена или сервера. Домен — это группа рабочих станций и серверов с общим групповым именем. Об именах почтовых ящиков мы расскажем далее в этой главе, когда будем подробно обсуждать реализацию простого клиента.

Поскольку для создания и передачи данных по сети почтовые ящики используют службы файловой системы Windows, их интерфейсный протокол независим. При создании приложения, процессы которого взаимодействуют по сети, не нужно знать, как работают нижележащие сетевые транспортные протоколы. Когда почтовые ящики устанавливают удаленное соединение с компьютерами в сети, для передачи данных от клиента к серверу перенаправитель Windows использует протокол Server Message Block (SMB). Сообщения обычно пересылаются без установления соединения, но в зависимости от размера сообщения вы можете заставить перенаправитель Windows устанавливать соединения на компьютерах с Windows NT или Windows 2000.

## Размеры сообщений

Для передачи сообщений по сети почтовые ящики обычно используют *дейтаграммы* (datagram) — небольшие порции данных, передаваемые по сети без установления соединения. Это ненадежный способ, поскольку уведомление о получении пакета данных не пересылается, и его доставка не гарантируется. Между тем, с помощью дейтаграмм можно передавать сообщения от одного клиента многим серверам. Этот механизм не работает на компьютерах с Windows NT и Windows 2000, если размер сообщения превышает 424 байта.

На компьютерах с Windows NT и Windows 2000 сообщения размером более 426 байт передаются с использованием протокола, требующего установления соединения в сеансе SMB, а не дейтаграммами. Это позволяет передавать большие сообщения надежно и эффективно. Впрочем, в этом случае вы теряете возможность передавать сообщение от одного клиента многим серверам. При установлении соединения допускается соединение только одного клиента с одним сервером и гарантируется доставка данных между процессами.

Однако интерфейс почтового ящика в Windows NT и Windows 2000 не гарантирует, что сообщение будет действительно записано в почтовый ящик. Например, если вы посылаете большое сообщение от клиента несуществующему серверу, интерфейс почтового ящика не сообщит клиентскому приложению, что не смог передать данные.

Поскольку Windows NT и Windows 2000 выбирают способ передачи в зависимости от размера сообщения, появляется проблема совместимости при передаче больших сообщений между компьютером с Windows NT или Windows 2000 и компьютером с Windows 95 или Windows 98.

Windows 95 и Windows 98 доставляют сообщения только посредством дейтаграмм, независимо от их размеров. Если клиент Windows 95 или Windows 98 попытается отправить сообщение больше 424 байт серверу Windows NT или Windows 2000, последний примет первые 424 байта и отбросит остальные, поскольку принимает большие сообщения в сеансе SMB с установлением соединения. Похожая проблема существует при передаче сообщений от клиента Windows NT или Windows 2000 серверу Windows 95 или Windows 98. Помните: Windows 95 и Windows 98 получают данные только посредством дейтаграмм. Поскольку Windows NT и Windows 2000 передают дейтаграммами только сообщения размером не более 426 байт, Windows 95 и Windows 98 не получают от таких клиентов сообщений большего размера (табл. 3-1)-

**Табл. 3-1. Ограничения размера сообщений почтовых ящиков**

Направление передачи	Передача посредством дейтаграмм без установления соединения	Передача с установлением соединения
Windows 95 или Windows 98 -> Windows 95 или Windows 98	Размер сообщения не более 64 кб	Не поддерживается
Windows NT или Windows 2000 -> Windows NT или Windows 2000	Размер сообщения не более 424 байт	Сообщения должны быть более 426 байт
Windows NT или Windows 2000 -> Windows 95 или Windows 98	Размер сообщения не более 424 байт	Не поддерживается
Windows 95 или Windows 98 -> Windows NT или Windows 2000	Размер сообщения не более 424 байт, иначе сообщение усекается до этого размера	Не поддерживается

*Vf*

**ПРИМЕЧАНИЕ** Windows CE не описана в табл. 3-1, потому что в этой ОС интерфейс программирования почтовых ящиков не доступен. Сообщения размером 425—426 байт также не описаны в таблице, из-за ограничений перенаправителей Windows NT и Windows 2000.

Перенаправители Windows NT и Windows 2000 не могут отправлять или принимать дейтаграммы размером 425—426 байт. Например, если вы отправляете от клиента Windows NT или Windows 2000 сообщение серверу Windows 95, Windows 98, Windows NT или Windows 2000, перед его отправкой серверу перенаправитель Windows NT усекает сообщение до 424 байт.

Чтобы обеспечить полную совместимость всех платформ Windows, ограничьте размер сообщений 424 байтами. При установлении соединений воспользуйтесь вместо почтовых ящиков именованными каналами.

## Компиляция приложения

При сборке клиента или сервера почтовых ящиков в Microsoft Visual C++ необходимо включать в программные файлы приложений заголовочный файл Winbase.h. Если вы включаете файл Windows.h (как в большинстве приложений), Winbase.h можно опустить. Ваше приложение также должно компоноваться с библиотекой Kernel32.lib, соответствующие параметры обычно настраивают с помощью флагов компоновщика Visual C++.

## Коды ошибок

Все API-функции Win32, используемые при разработке клиентов и серверов почтовых ящиков (за исключением *CreateFile* и *CreateMailslot*), в случае неудачи возвращают 0. API-функции *CreateFile* и *CreateMailslot* возвращают значение *INVALID\_HANDLE\_VALUE*. Для получения кодов ошибок этих функций, приложение должно вызывать функцию *GetLastError*. Полный список кодов ошибок см. в приложении С или в файле Winerror.h.

## Общие сведения об архитектуре клиент-сервер

Почтовые ящики используют простую архитектуру клиент-сервер, в которой данные передаются от клиента серверу однонаправленно. Сервер отвечает за создание почтового ящика и является единственным процессом, который может читать из него данные. Клиенты почтовых ящиков — это процессы, открывающие экземпляры почтовых ящиков и единственные, имеющие право записывать в них данные.

## Сервер почтовых ящиков

Для реализации почтового ящика нужно написать базовое серверное приложение, в котором выполнить следующие действия.

- Создать описатель почтового ящика с помощью API-функции *CreateMailslot*.  
Получить данные от любого клиента путем вызова API-функции *ReadFile* с описателем почтового ящика в качестве параметра.  
Закрыть описатель почтового ящика с помощью API-функции *CloseHandle*.



Как видно, для разработки серверного приложения почтового ящика требуется совсем немного вызовов API-функций.

Серверные процессы создают почтовые ящики с помощью вызова API-функции *CreateMailslot*, которая определена так:

```
HANDLE CreateMailslot(  
    LPCSTR IpName,  
    DWORD nMaxMessageSize,  
    DWORD ReadTimeout,  
    LPSECURITY_ATTRIBUTES IpSecurityAttributes
```

Параметр *IpName* задает **имя почтового ящика**. Оно должно иметь следующий ВИД:

```
\\.\Mailslot\[путь]имя
```

Заметьте: имя сервера представлено точкой, что означает локальный компьютер. Это необходимо, поскольку нельзя создать почтовый ящик на удаленном компьютере. В параметре *IpName* имя обязательно уникально, но может быть простым или включать полный путь.

Параметр *nMaxMessageSize* задает максимальный размер (в байтах) сообщения, которое может быть записано в почтовый ящик. Если клиент записывает сообщение большего размера, сервер не видит это сообщение. Если задать значение 0, то сервер будет принимать сообщения любого размера.

Операции чтения (подробнее — далее в этой главе) могут выполняться с блокировкой почтового ящика или без таковой, в зависимости от параметра *IReadTimeout*, задающего количество времени (в миллисекундах), в течение которого операции чтения ждут входящих сообщений. Значение *MAILSLOT\_WAIT\_FOREVER* позволит заблокировать операции чтения и заставит их бесконечно ожидать, пока входящие данные не станут доступны для чтения. Если задать значение 0, операции чтения возвращаются немедленно.

Параметр *IpSecurityAttributes* определяет права доступа к почтовому ящику. В Windows 95 и Windows 98 он должен иметь значение *NULL*, потому что в этих ОС система безопасности к объектам не применима. В Windows NT и Windows 2000 этот параметр реализован частично, поэтому следует также присвоить ему *NULL*.

Почтовый ящик относительно безопасен только при локальном вводе-выводе, когда клиент пытается открыть этот ящик, используя точку (.) в качестве имени сервера. Клиент может обойти систему безопасности, задав вместо точки фактическое имя сервера, как при удаленном вызове ввода-вывода. Параметр *IpSecurityAttributes* не реализован для удаленного ввода-вывода в Windows NT и Windows 2000 из-за больших издержек, связанных с формированием аутентифицированного сеанса между клиентом и сервером при каждой отправке сообщения. Таким образом, почтовые ящики только частично соответствуют модели безопасности Windows NT и Windows 2000, реализованной в стандартных файловых системах. В итоге, любой клиент почтового ящика в сети может отправлять данные серверу.

После создания почтового ящика с действительным описателем можно читать данные. Сервер — единственный процесс, который может читать данные из почтового ящика. Для он должен вызвать "№т32-функцию *ReadFile*, определенную так:

```
BOOL ReadFile(
    . HANDLE hFile,
      LPVOID lpBuffer,
      DWORD nNumberOfBytesToRead,
      LPDWORD lpNumberOfBytesRead,
      LPOVERLAPPED lpOverlapped
);
```

*CreateMailslot* возвращает описатель *hFile*. Параметры *lpBuffer* и *nNumberOfBytesToRead* определяют, сколько данных может быть считано из почтового ящика. Важно задать размер этого буфера большим, чем параметр *nMaxMessageSize* из API-вызова *CreateMailslot*. Кроме того, размер буфера должен превышать размеры входящих сообщений, иначе *ReadFile* выдаст код ошибки *ERROR\_INSUFFICIENT\_BUFFER*. Параметр *lpNumberOfBytesRead* возвращает количество считанных байтов по завершении работы *ReadFile*.

Параметр *lpOverlapped* позволяет считывать данные из почтового ящика асинхронно. Он использует Win32-Механизм *перекрытого ввода-вывода* (overlapped I/O), подробно описанный в главе 4. По умолчанию, выполнение *ReadFile* блокирует (ожидает) ввод-вывод, пока данные не станут доступны для чтения. Перекрытый ввод-вывод применим только в Windows NT и Windows 2000, в Windows 95 или Windows 98 присвойте параметру *lpOverlapped* значение *NULL*. В листинге 3-1 показано, как написать простой сервер почтовых ящиков.

Листинг 3-1. Пример сервера почтовых ящиков

```
// Served.cpp

<include <windows.h>
<include <stdio.h>

void main(void)
{
    HANDLE Mailslot;
    char buffer[256];
    DWORD NumberOfBytesRead;

    // Создание почтового ящика
    if ((Mailslot = CreateMailslot("\\\\V\\Mailslot\\Myslot", 0,
        MAILSLT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a mailslot Xd\\n", GetLastError());
    }
}
```

см.след.стр.

**Листинг 3-1. (продолжение)**

```

// Бесконечное чтение данных из почтового ящика
while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead,
    NULL) != 0)
{
    printf( X "%s\n", NumberOfBytesRead, buffer),

```

**Клиент почтовых ящиков**

Для реализации клиента нужно разработать приложение, ссылающееся на существующий почтовый ящик и записывающее в него данные. В базовом клиентском приложении необходимо выполнить следующие шаги:

- 1 Открыть описатель-ссылку на почтовый ящик, в который нужно отправить данные, с помощью API-функции *CreateFile*
- 2 Записать данные в почтовый ящик, вызвав API-функцию *WriteFile*
- 3 Закрыть описатель почтового ящика с помощью API-функции *CloseHandle*

Как уже говорилось, клиенты почтовых ящиков соединяются с серверами без установления соединения. Когда клиент открывает описатель-ссылку на почтовый ящик, он не устанавливает связь с сервером почтового ящика. На почтовые ящики ссылаются путем вызова API-функции *CreateFile*, определенной так:

```

HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
),

```

Параметр *lpFileName* описывает один или несколько почтовых ящиков, в которые можно поместить данные. Для этого укажите имя ящика в одном из форматов, описанных в этой главе. Правила именования почтовых ящиков таковы:

**I** \\.\mailslot\<имя> — определяет локальный почтовый ящик на том же компьютере,

**II** \\<имя\_сервера>\mailslot\<имя> — определяет удаленный сервер почтового ящика с именем *имя\_сервера*,

**III** \\<имя\_домена>\<имя> — определяет все почтовые ящики с именем *имя* в домене *имя\_домена*,

**IV** \\.\mailslot\<имя> — определяет все почтовые ящики с именем *имя* в основном домене системы.

Параметр *dwDestredAccess* должен иметь значение *GENERICWRITE*, потому что клиент может только записывать данные на сервер. Параметр *dwShareMode* обязан иметь значение *FILE\_SHARE\_READ*, позволяя серверу открывать и выполнять операции чтения из почтового ящика. Значение параметра *ipSecurityAttributes* не влияет на почтовые ящики — следует задать *ALL*. Флаг *dwCreationDisposition* должен быть равен *OPEN\_EXISTING*. Это удобно, когда клиент и сервер функционируют на одном и том же компьютере. Если сервер не создал почтовый ящик, API-функция *CreateFile* вернет ошибку. Параметр *dwCreationDisposition* не имеет значения, если сервер работает удаленно. Параметр *dwFlagsAndAttributes* должен иметь значение *FILE\_ATTRIBUTE\_NORMAL*, а *hTemplateFile* — значение *NULL*.

После успешного создания описателя можно помещать данные в почтовый ящик. Помните, что клиент может только записывать данные в почтовый ящик с помощью *Win32^уНК4НН WntFile*.

```
BOOL WntFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWnt,
    LPDWORD lpNumberOfBytesWntten,
    LPOVERLAPPED lpOverlapped
),
```

Параметр *hFile* — это описатель-ссылка, возвращаемый функцией *CreateFile*. Параметры *lpBuffer* и *nNumberOfBytesToWnt* определяют, сколько байт будет отправлено от клиента серверу. Максимальный размер сообщения — 64 кб. Если описатель почтового ящика создан с указанием домена или звездочки, размер сообщения не должен превышать 424 байта в Windows NT и Windows 2000, или 64 кб — Windows 95 и Windows 98. Если клиент попытается отправить сообщение большего размера, функция *WntFile* вернет ошибку *ERROR\_BAD\_NETPATH* (чтобы узнать код ошибки, вызовите функцию *GetLastError*). Это происходит потому, что сообщение посылается всем серверам сети как широковещательная дeйтаграмма. Параметр *ipNumberOfBytesWritten* возвращает количество байт, отправленных серверу после завершения функции *WntFile*.

Параметр *lpOverlapped* позволяет записывать данные в почтовый ящик асинхронно. Поскольку почтовые ящики обмениваются данными без установления соединения, функция *WntFile* не блокирует ввод-вывод. На клиенте этот параметр должен быть равен *NULL*. В листинге 3-2 приведен пример простого клиента почтового ящика.

Листинг 3-2. Пример клиента почтового ящика

// Client.cpp

```
"include <windows.h>
<include <stdio.h>
```

```
m n ( i n t argc, char .argv[]>
```

ш с/ид

**Листинг 3-2. {продолжение}**

```

{
    HANDLE Mailslot;
    DWORD BytesWritten;
    CHAR ServerName[256];

    // Ввод аргумента командной строки для сервера, которому отправляется сообщение
    if (argc < 2)
    {
        printf("Usage: client <server name>\n");
        return;
    }
    sprintf(ServerName, "\\\\"Xs\\Mailslot\\Myslot", argv[1]);

    if ((Mailslot = CreateFile(ServerName, GENERIC_WRITE,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error Xd\n", GetLastErrorO);
        return;
    }

    if (WriteFile(Mailslot, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error Xd\n", GetLastErrorO);
        return;
    }

    printf("Wrote Xd bytes\n", BytesWritten);

    CloseHandle(Mailslot);
}

```

## Дополнительные API-функции ПОЧТОВЫХ ЯЩИКОВ

Сервер почтового ящика может использовать две дополнительные API-функции для взаимодействия с почтовым ящиком: *GetMailslotInfo* и *SetMailslotInfo*. Функция *GetMailslotInfo* возвращает размер сообщения, когда оно прибывает в почтовый ящик. Приложения используют эту возможность, чтобы динамически настраивать свои буферы для входящих сообщений изменяющейся длины. *GetMailslotInfo* может также применяться для опроса наличия входящих данных:

```

BOOL GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpCurrentSize,
    LPDWORD lpWaitTimeOut)

```

```
LPDWORD IpMessageCount,
LPDWORD IpReadTimeout
```

```
);
```

Параметр *hMailslot* указывает почтовый ящик, возвращенный вызовом др!\_функции *CreateMailslot*. Параметр *ipMaxMessageSize* задает, сообщение какого размера (в байтах) можно записать в почтовый ящик. Параметр *IpNextSize* указывает на размер следующего сообщения (в байтах). Параметр *GetMailslotInfo* может вернуть значение *MAILSLOT\_NO\_MESSAGE*, указывая, что в настоящий момент почтовый ящик не ждет никакого сообщения. Потенциально сервер вправе использовать этот параметр для проверки наличия входящих данных, не давая приложению блокировать ввод-вывод при вызове функции *ReadFile*. Но делать это не рекомендуется: приложение будет непрерывно использовать центральный процессор для проверки входящих данных, даже если не обрабатываются никакие сообщения, что уменьшит производительность.

Если вы хотите предотвратить блокирование при вызове функции *ReadFile*, используйте перекрытый ввод-вывод Win32. Параметр *IpMesssageCount* задает буфер, куда записывается общее количество сообщений, ожидающих прочтения. Этот параметр также задействуют для проверки наличия данных. Параметр *IpReadTimeout* указывает на буфер, возвращающий время тайм-аута (в миллисекундах), в течение которого операция чтения ждет записи сообщения в почтовый ящик.

API-функция *SetMailslotInfo* задает значение тайм-аута для почтового ящика, в течение которого операция чтения ожидает входящих сообщений. Таким образом, приложение может изменить способ чтения от блокирующего к неблокирующему или наоборот. Параметр *SetMailslotInfo* определен так:

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD IReadTimeout
);
```

Параметр *hMailslot* указывает почтовый ящик, возвращаемый вызовом API-функции *CreateMailslot*. Параметр *IReadTimeout* определяет количество времени (в миллисекундах), в течение которого операция чтения ожидает записи сообщения в почтовый ящик. Если оно равно 0, то в отсутствие сообщений операции чтения возвращаются немедленно, если *MAILSLOT\_WAIT\_FOREVER* — будут ждать бесконечно долго.

## Платформа и производительность

Почтовые ящики в Windows 95 и Windows 98 имеют ограничения: они именуются по правилу «8.3», не позволяют отменить блокирующие запросы ввода-вывода, вызывают утечки памяти по истечении тайм-аута.

### Правила именования «8.3»

Windows 95 и Windows 98 ограничивают размеры имен почтовых ящиков форматом «8.3». Это создает проблемы совместимости между Windows 95

или Windows 98 и Windows NT или Windows 2000. Например, если вы попытаетесь создать или открыть почтовый ящик с именем \\ \Mailslot\Mymailslot, Windows 95 создаст почтовый ящик \\ \Mailslot\Mymailsl и будет ссылаться на него. Функции *CreateMailslot* и *CreateFile* выполняются успешно, несмотря на усеменение имени. Если затем вы отправите сообщение от Windows 2000 к Windows 95 или наоборот, оно не будет получено из-за несоответствия имен почтовых ящиков. Если клиент и сервер работают на компьютерах с Windows 95, проблем не возникает — имя усекается как на клиенте, так и на сервере. Чтобы избежать осложнений, не создавайте почтовые ящики с длинными именами.

## Неспособность отменить блокирующие запросы ввода-вывода

Эта проблема существует в Windows 95 и Windows 98. Серверы почтовых ящиков для получения данных вызывают функцию *ReadFile*. Если почтовый ящик создается с флагом *MAILSLOT\_WAIT\_FOREVER*, запросы блокируются на неопределенное время, пока данные не станут доступны. При невыполненном запросе функции *ReadFile* серверное приложение при завершении зависает. Единственный способ снять приложение — перезагрузить Windows. Для решения этой проблемы заставьте сервер открыть дескриптор его почтового ящика в отдельном потоке и отправить данные, чтобы прервать блокирующий запрос чтения (см. листинг 3-3).

### Листинг 3-3. Исправленный сервер почтовых ящиков

// **Server2.cpp**

```
((include <windows.h>
<include <stdio.h>
<include <conio.h>

BOOL StopProcessmg,

DWORD WINAPI ServeMailslot(LPVOID lpParameter),
void SendMessageToMailslot(void),

void raam(void) {

    DWORD ThreadId,
    HANDLE MailslotThread,

    StopProcessmg = FALSE,
    MailslotThread = CreateThread(NULL, 0, ServeMailslot, NULL,
        0, &ThreadId),

        i Mi) itji

    printf( Press a key to stop the server\n );_, "
    _getch(), B.8» I*

    // Флаг StopProcessmg присваивается TRUE/«ТОМ при завершении
```

## Листинг 3-3. (продолжение)

```

    Ц функции ReadFile поток сервера закончился
    StopProcessing = TRUE,

    // Отправка сообщения почтовому ящику, чтобы прервать
    // вызов функции ReadFile на сервере
    ** SendMessageToMailslot0,

    // Ожидание завершения выполнения потока сервера
    if (WaitForSingleObject(MailslotThread, INFINITE) == WAIT_FAILED)
    {
        printf( WaitForSingleObject failed with error %d\n ,
                GetLastErrorO),
        return,

// Функция ServeMailslot
//
// Описание
// Эта рабочая функция сервера почтового ящика
// для обработки всего входящего ввода-вывода ящика
//
DWORD WINAPI ServeMailslot(LPVOID lpParameter)
{
    char buffer[2048],
    DWORD NumberOfBytesRead,
    DWORD Ret,
    HANDLE Mailslot,

    if ((Mailslot = CreateMailslot( \\.\mailslot\myslof, 2048,
        MAILSLT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf( Failed to create a MailSlot %d\n , GetLastErrorO);
        return 0,

    while((Ret = ReadFile(Mailslot, buffer, 2048,
        &NumberOfBytesRead, NULL)) != 0)
    {
        if (StopProcessing)
            break,

        printf( Received %d bytes\n , NumberOfBytesRead);

    CloseHandle(Mailslot),

```



## Листинг 3-2. (продолжение)

```

return 0;
>

// Функция SendMessageToMailslot
//
// Описание.
// Функция SendMessageToMailslot отправляет простое сообщение
// серверу, чтобы прервать блокирующий вызов API-функции ReadFile
//
void SendMessageToMailslot(void)
{
    HANDLE Mailslot;
    DWORD BytesWritten;

    if ((Mailslot = CreateFile("\\\\.\\mailslot\\inyslot",
        GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastErrorO);
        return;
    }

    if (WriteFile(Mailslot, "STOP", 4, &BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastErrorO);
        return;
    }

    CloseHandle(Mailslot);
}

```

/N)1019lf<

## Утечки памяти

Утечки памяти в Windows 95 и Windows 98 происходят при использовании значений тайм-аута в почтовых ящиках. Если почтовый ящик создается функцией *CreateMailslot* со значением тайм-аута, большим 0, функция *ReadFile* вызывает утечку памяти, когда время ожидания истекает, и функция возвращает *FALSE*. После многократных вызовов функции *ReadFile* система становится нестабильной и последующие вызовы этой функции, время ожидания которых истекает, начинают возвращать *TRUE*. В результате система больше не может выполнять другие MS-DOS-приложения. Для решения этой проблемы создавайте почтовый ящик со временем ожидания, равным 0 или *MAILSLOT\_WAIT\_FOREVER*. Это не позволит приложению использовать механизм тайм-аутов, вызывающий утечки памяти.

В базе знаний Microsoft, к которой можно обратиться по адресу <http://support.microsoft.com/support/search>, описаны следующие проблемы и ограничения.

**Я Q139715** — функция *ReadFile* возвращает неверный код ошибки для почтовых ящиков. Когда сервер открывает почтовый ящик функцией *CreateMmlslot*, задает тайм-аут и затем получает данные функцией *ReadFile*, последняя выдает ошибку 5 (доступ отклонен), если данные недоступны.

**III Q192276** — функция *GetMailslotInfo* возвращает неверное значение параметра *ipNextSize*. Если вызвать эту API-функцию в Windows 95 OEM Service Release 2 (OSR2) или Windows 98 без установки компонента сетевого клиента, вы получите неправильное значение (обычно в миллионах) или отрицательное число для параметра *IpNextSize*. Если вызвать функцию повторно, она обычно начинает возвращать правильное значение.

- **Q170581** — почтовый ящик, созданный в Windows 95, вмещает только 4093 байта. Вызов API-функции *WriteFile* для записи более 4093 байт в почтовый ящик, созданный на рабочей станции Windows 95, не завершается успешно.

**III Q131493** — функция *CreateFile* и почтовые ящики. В документации на API-функцию *CreateFile* неверно описаны возможные значения, возвращаемые ею при открытии клиентской части почтового ящика

## Резюме

Сетевая технология почтовых ящиков предоставляет приложениям простой однонаправленный механизм межпроцессной связи с использованием перенаправителя Windows. Наиболее полезная возможность — широковещательная передача сообщений одному или нескольким компьютерам в сети. Впрочем, почтовые ящики не обеспечивают надежную передачу данных.

Если вы хотите надежно передавать данные с помощью перенаправителя Windows, используйте именованные каналы, которым посвящена следующая глава.

## Именованные каналы

Именованные каналы — это простой механизм *связи между процессами* (interprocess communication, IPC), поддерживаемый в Microsoft Windows NT, Windows 2000, Windows 95 и Windows 98 (но не Windows CE) Именованные каналы обеспечивают надежную одностороннюю и двустороннюю передачу данных между процессами на одном или разных компьютерах Разработка приложений, работающих с именованными каналами, не представляет сложности и не требует особых знаний механизма работы основных сетевых протоколов (таких как TCP/IP или IPX) Детали работы протоколов скрыты от приложения, так как для обмена данными между процессами через сеть именованные каналы используют перенаправитель сети Microsoft — Microsoft Network Provider (MSNP) Очень важно и то, что именованные каналы позволяют воспользоваться встроенными возможностями защиты Windows NT или Windows 2000

К примерам использования именованных каналов можно отнести разработку системы управления данными, которая позволяет выполнять транзакции только определенной группе пользователей Рассмотрим следующую ситуацию в офисе есть компьютер с некоей секретной информацией, доступ к которой должен иметь только управленческий персонал Допустим, каждый сотрудник фирмы должен видеть этот компьютер со своей рабочей станции, однако простые служащие не вправе иметь доступ к конфиденциальным данным Эта проблема решается с помощью именованных каналов Можно разработать серверное приложение, которое будет выполнять транзакции над секретными данными в зависимости от запросов клиентов С помощью встроенных возможностей защиты Windows NT или Windows 2000 сервер ограничит доступ рядовых сотрудников к конфиденциальным данным

Именованные каналы представляют собой простую архитектуру клиент-сервер, обеспечивающую надежную передачу данных Эта глава посвящена разработке сервера и клиента именованного канала Мы начнем с обсуждения названий и типов простых именованных каналов Затем создадим простой сервер и рассмотрим детали реализации более сложного сервера После этого разберем создание простого клиента именованного канала В конце главы будут перечислены основные проблемы и ограничения, связанные с именованными каналами

## Детали реализации именованных каналов

Для работы с файловой системой Windows именованные каналы используют интерфейс Named Pipe File System (NPFS). Для получения и отправки данных сервер и клиент применяют стандартные API-функции Win32, такие как *ReadFile* и *WriteFile*. Эти функции позволяют приложениям использовать правила именования файловой системы Win32 и возможности защиты Windows NT и Windows 2000. При отправке и получении данных по сети интерфейс NPFS задействует перенаправитель MSNP. Это делает его независимым от протокола: при разработке приложений программист не заботится о деталях работы протоколов (TCP/IP или IPX). Названия именованных каналов должны удовлетворять формату Universal Naming Convention (UNC). Подробнее о UNC, перенаправителе Windows и безопасности — в главе 2.

### Правила именования каналов

Имена каналов имеют следующий формат

`\\сервер\Pipe\[путь]имя`

Данная строка состоит из трех частей: `\\сервер`, `\Pipe` и `[путь]имя`, где `\\сервер` — имя сервера, на котором создан именованный канал, `\Pipe` — фиксированная обязательная строка, уведомляющая систему о принадлежности к NPFS, `[путь]имя` — уникальное имя канала, включающее несколько уровней каталогов. Вот примеры правильных названий именованных каналов

```
\\myserver\Pipe\mypipe
\\Testserver\pipe\cooldirectory\funtest\jim
\\ \Pipe\Easynamedpipe
```

Имя сервера может быть представлено точкой

### Режимы побайтовый и сообщений

Именованные каналы используют два режима передачи данных: побайтовый и сообщений. В первом случае сообщения передаются непрерывным потоком байтов между клиентом и сервером. Это означает, что клиент и сервер точно не знают, сколько байтов считывается или записывается в канал в определенный момент времени. Таким образом, запись одного количества байтов не означает чтение того же количества с другой стороны канала. Такой способ передачи позволяет клиенту и серверу не заботиться о содержимом передаваемых данных. Во втором случае клиент и сервер отправляют и принимают данные дискретными блоками, при этом каждое сообщение прочитывается целиком (рис. 4-1).

### Компиляция приложений

При создании клиента или сервера именованного канала с помощью Microsoft Visual C++ необходимо включить в программные файлы заголовочный

файл `Winbase.h`. Если приложение включает файл `Windows.h` (как правило, это так), файл `Winbase.h` можно опустить. Кроме того, как уже упоминалось, с помощью флагов компоновщика Visual C++ необходимо подключить библиотеку `Kernel32.lib`.

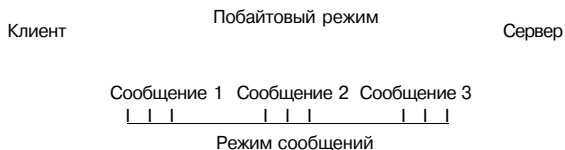


Рис. 4-1. Режимы побайтовый и сообщений

## Коды ошибок

Все API-функции Win32 (кроме *CreateFile* и *CreateNamedPipe*), используемые при разработке сервера и клиента именованного канала, при возникновении ошибки возвращают значение 0. Функции *CreateFile* и *CreateNamedPipe* возвращают значение *INVALID\_HANDLE\_VALUE*. Подробную информацию об ошибке можно получить с помощью функции *GetLastError*. Полный список кодов ошибок — в файле `Winerror.h` или в приложении В.

## Простой сервер и клиент

Именованные каналы имеют простую архитектуру клиент-сервер, при которой данные передаются в одном или двух направлениях. Это позволяет и серверу, и клиенту отправлять и принимать данные. Основное отличие сервера от клиента — только сервер может создать именованный канал и принять соединение с клиентом. Клиентское приложение устанавливает соединение с существующим сервером. После этого сервер и клиент могут читать и записывать данные в канал с помощью стандартных API-функций Win32, таких как *ReadFile* и *WriteFile*. Сервер именованного канала работает только на компьютере с Windows NT или Windows 2000, Windows 95 и Windows 98 не поддерживают создание именованных каналов. Это ограничение не позволяет двум компьютерам Windows 95 или Windows 98 напрямую обмениваться данными, однако их клиенты могут подключаться к серверам на компьютерах Windows NT и Windows 2000.

## Детали реализации сервера

Реализация сервера именованного канала подразумевает разработку приложения, создающего экземпляры каналов, к которым подключаются клиенты.

Для сервера экземпляр именованного канала — это просто описатель, с помощью которого устанавливается соединение с локальными или удаленными клиентами. Процесс создания простого сервера заключается в последовательном использовании API-функций:

- III **CreateNamedPipe** — для создания экземпляра именованного канала;
- S **ConnectNamedPipe** — для прослушивания клиентских соединений;
- III **ReadFile** и **WriteFile** — для получения и отправки данных;
- III **DisconnectNamedPipe** — для завершения соединения;
- III **CloseHandle** — для закрытия описателя экземпляра именованного канала.

Сначала сервер должен создать экземпляр именованного канала с помощью API-функции *CreateNamedPipe*:

```
HANDLE CreateNamedPipe(
    LPCSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Параметр *lpName* определяет название именованного канала, удовлетворяющее формату UNQ

\\.\Pipe\[путь]имя

Имя сервера представлено точкой — значит, в этом качестве используется локальный компьютер. Именованный канал нельзя создать на удаленном компьютере. Часть параметра */путь/имя* определяет уникальное имя канала. Это может быть просто имя файла или полный путь к нему.

Параметр *dwOpenMode* определяет направление передачи, управление вводом-выводом и безопасность канала. В табл. 4-1 перечислены флаги, комбинации которых используют при создании канала.

**Табл. 4-1. Флаги режимов создания именованного канала**

Режим открытия	Флаг	Описание
Направленный	<i>PIPE_ACCESS_DUPLEX</i>	Канал двунаправленный: серверные и клиентские процессы могут принимать и отправлять данные по каналу
	<i>PIPE_ACCESS_OUTBOUND</i>	Данные передаются только от сервера к клиенту
	<i>PIPE_ACCESS_INBOUND</i>	Данные передаются только от клиента к серверу

см. след. стр.

Табл. 4-1. (продолжение)

Режим открытия	Флаг	Описание
Управление вводом- выводом	<i>FILEFLAGWRITE_</i> <i>THROUGH</i>	Функции записи не возвращают значение, пока данные передаются по сети или находятся в буфере удаленного компьютера. Применяется только для именованных каналов, работающих в побайтовом режиме
	<i>FILEFLAGOVERLAPPED</i>	Позволяет использовать перекрытый ввод-вывод при выполнении операций чтения, записи и соединения
Безопасность	<i>WRITE_DAC</i>	Позволяет приложению изменять список <i>DACL</i> именованного канала
	<i>ACCESSSYSTEM^SECURITY</i>	Позволяет приложению изменять список <i>SACL</i> именованного канала
	<i>WRITE_OWNER</i>	Позволяет приложению изменять владельца именованного канала и групповой <i>SID</i>

Флаги *PIPE\_ACCESS\_* определяют направление передачи данных между сервером и клиентом. Если открыть канал с флагом *PIPE^ACCESS\_DUPLEX*, передача по нему будет двунаправленной. При открытии канала с флагом *PIPE\_ACCESS\_INBOUND* или *PIPE\_ACCESS\_OUTBOUND* — однонаправленной: данные будут передаваться только от клиента серверу или наоборот. На рис. 4-2 изображены комбинации флагов и направление передачи между сервером и клиентом.

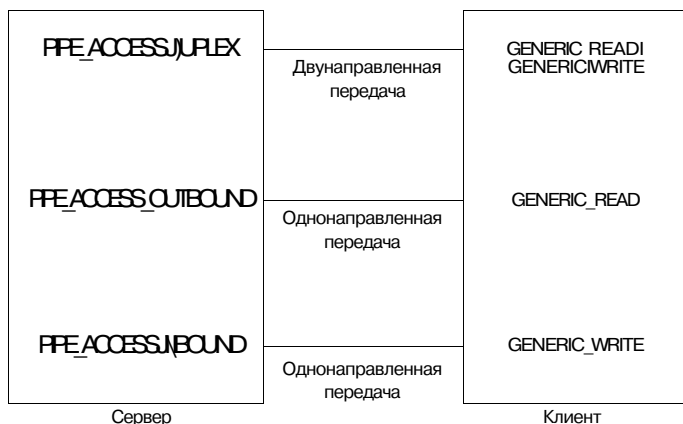


Рис. 4-2. Флаги режимов и направление передачи

Следующий набор флагов управляет операциями ввода-вывода сервера. При применении флага *FILE\_FLAG\_WRITE\_THROUGH* функции записи не возвращают значение, пока данные не будут переданы по сети или не появятся в буфере удаленного компьютера. Этот флаг используют, когда сервер и клиент находятся на разных компьютерах и именованный канал работает в по-

байтовом режиме. Флаг *FILEFLAG^OVERLAPPED* позволяет функциям, выполняющим операции чтения, записи и соединения, немедленно возвращать значение, даже если их выполнение требует существенных затрат времени. Перекрытый ввод-вывод мы подробно обсудим в разделе, посвященном разработке более сложного сервера.

Последняя группа флагов — *dwOpenMode* (табл. 4-1), управляет доступом сервера к дескриптору безопасности, созданному именованным каналом. Их используют, если приложению требуется обновить или изменить дескриптор после создания канала. Флаг *WRITE\_DAC* позволяет приложению обновлять *список избирательного управления доступом* (discretionary access control list, DACL); флаг *ACCESS\_SYSTEM\_SECURITY* — получить доступ к *системному списку управления доступом* (system access control list, SACL); флаг *WRITEOWNER* — изменять владельца именованного канала и *групповой идентификатор безопасности* (security ID, SID). Например, можно запретить доступ пользователя к именованному каналу, изменив список DACL канала с помощью API-функций. Подробнее о DACL, SACL и SID — в главе 2.

Параметр *dwPipeMode* определяет режимы операций чтения, записи и ожидания. При создании канала нужно указать по одному флагу из каждой категории, объединив их с помощью операции OR (табл. 4-2).

**Табл. 4-2. Флаги режимов чтения-записи именованного канала**

Режим открытия	Флаг	Описание
Запись	<i>PIPE_TYPE_BYTE</i>	Данные записываются в канал потоком байтов
	<i>PIPE_TYPE_MESSAGE</i>	Данные записываются в канал потоком сообщений
Чтение	<i>PIPE_READMODE_BYTE</i>	Данные считываются из канала потоком байтов
	<i>PIPE_READMODE_MESSAGE</i>	Данные считываются из канала потоком сообщений
Ожидание	<i>PIPEWAIT</i>	Включен режим блокировки
	<i>PIPENOWAIT</i>	Отключен режим блокировки

Если при создании побайтового канала использовались флаги *PIPE\_READMODE|PIPE\_TYPE\_BYTE*, то данные будут записываться и считываться из канала только потоком байтов. В этом случае не обязательно уравнивать количество операций чтения и записи, поскольку данные не разделяются на отдельные сообщения. Например, если в канал записано 500 байт, получатель может считывать по 100 байт, до тех пор пока не считает все данные.

Чтобы сообщения имели четкие границы, создайте канал в режиме сообщений, указав флаги *PIPE\_READMODE\_MESSAGE|PIPE\_TYPE\_MESSAGE*. В этом случае необходимо уравнивать количество операций чтения и записи данных. Например, если в канал записано 500 байт, то для чтения данных Потребуется буфер размером 500 байт или более, иначе функция *ReadFile* выдаст ошибку *ERROR\_MORE\_DATA*, Флаг *PIPE\_TYPE\_MESSAGE* можно комбинировать с флагом *PIPE\_READMODE\_BYTE*. Это позволит отправлять данные



в режиме сообщений, а при их получении — считывать произвольное количество байтов. При таком способе передачи разделители сообщений игнорируются. Флаг *PIPEJTYPE BYTE* нельзя использовать с флагом *PIPEREADMODE\_MESSAGE*, иначе функция *CreateNamedPipe* выдаст ошибку *ERROR\_INVALID\_PARAMETER*, поскольку при таком способе записи поток данных не будет содержать разделителей сообщений.

Флаги *PIPE\_WAIT* и *PIPE\_NOWAIT* переводят канал в блокирующий и неблокирующий режим соответственно. Их можно комбинировать с флагами режима чтения и записи. В режиме блокировки такие операции ввода-вывода, как *ReadFile*, блокируются, до тех пор пока запрос ввода-вывода не будет выполнен полностью. Такое поведение является стандартным, когда не указано ни одного флага. Если режим блокировки отключен, операции ввода-вывода возвращают значение немедленно. Однако данный режим не следует использовать для достижения асинхронного выполнения операций ввода-вывода в приложениях Win32. Он обеспечивает лишь совместимость с ранними версиями Microsoft LAN Manager 2.0. Асинхронное выполнение функций *ReadFile* и *WriteFile* достигается с помощью перекрытого ввода-вывода.

Параметр *nMaxInstances* определяет максимальное количество экземпляров (описателей) канала, которые одновременно может создать сервер. Экземпляр канала — это соединение локального или удаленного клиента с сервером именованного канала. Параметр может принимать значения от 1 до *PIPEJNLIMITEDINSTANCES*. Например, если сервер должен одновременно поддерживать до пяти соединений, присвойте параметру значение 5. Если параметр равен *PIPEJNLIMITEDINSTANCES*, количество экземпляров канала ограничено только системными ресурсами.

Параметры *nOutBufferSize* и *nInBufferSize* функции *CreateNamedPipe* определяют размеры входящего и исходящего внутренних буферов. При создании экземпляра канала система каждый раз формирует входящий и (или) исходящий буфер, задействуя резидентный пул (физическая память, используемая операционной системой). Размер буфера должен быть рациональным: не слишком большим, чтобы система не исчерпала резидентный пул, и не слишком малым, чтобы вместить стандартные запросы ввода-вывода. При попытке записать данные большего размера, система попытается автоматически расширить объем буфера, используя резидентный пул. Оптимальные размеры — те, которые приложение использует при вызове функций *ReadFile* и *WriteFile*.

Параметр *nDefaultTimeOut* задает стандартный тайм-аут (время ожидания соединения) в миллисекундах. Действие параметра распространяется только на клиентские приложения, определяющие, можно ли установить соединение с сервером с помощью функции *WaitNamedPipe* (подробней — далее в этой главе).

Параметр *ipSecurityAttributes* позволяет приложению указать дескриптор безопасности именованного канала и определяет, сможет ли дочерний процесс наследовать созданный описатель. Если этот параметр равен *NULL*, именованный канал использует стандартный дескриптор безопасности, а описатель не может быть унаследован. Стандартный дескриптор безопасности

подразумевает, что именованный канал имеет те же права доступа, что и создавший его процесс (см. модель безопасности Windows NT и Windows 2000 в главе 2). Приложение может предоставить доступ к каналу определенным пользователям и группам, определив структуру *SECURITY\_DESCRIPTOR* с помощью API-функций. Чтобы открыть доступ к серверу любым клиентам, в структуре *SECURITY\_DESCRIPTOR* следует задать пустой список DACL.

Когда функция *CreateNamedPipe* вернет описатель именованного канала, сервер начинает ждать соединения клиентов. Для установления соединения предназначена функция *ConnectNamedPipe*, которая определена так:

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
```

Параметр *hNamedPipe* — это описатель экземпляра именованного канала, возвращенный функцией *CreateNamedPipe*. Если при создании канала использовался флаг *FILE\_FLAG\_OVERLAPPED*, параметр *lpOverlapped* позволяет *ConnectNamedPipe* выполняться асинхронно, то есть без блокировки. Если этот параметр равен *NULL*, *ConnectNamedPipe* блокируется, до тех пор пока клиент не установит соединение с сервером.

Вызов функции *ConnectNamedPipe* завершается после установления соединения. Далее сервер отправляет и принимает данные с помощью API-функций *WriteFile* и *ReadFile*. Когда обмен данными с клиентом завершен, сервер должен закрыть соединение, вызвав функцию *DisconnectNamedPipe*. В листинге 4-1 показано, как написать простой сервер, способный обмениваться данными с одним клиентом.

#### Листинг 4-1. Простой сервер именованного канала

```
// Server.cpp

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

void main(void)

    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];
    if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1,
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)

        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return;
}
```

см. *cned.cmp*.

**Листинг 4-1.** *{продолжение}*

```

printf( Server is now running\n ),

if (ConnectNamedPipe(PipeHandle, NULL) == 0)
{
    printf( ConnectNamedPipe failed with error Xd\n
           GetLastErrorO),
    CloseHandle(PipeHandle),
    return,

if (ReadFile(PipeHandle, buffer, sizeof(buffer),
             &BytesRead, NULL) <= 0)
{
    printf( ReadFile failed with error Xd\n , GetLastErrorO),
    CloseHandle(PipeHandle),
    return,

printf( % *s\n', BytesRead, buffer),

if (DisconnectNamedPipe(PipeHandle) == 0)
{
    printf( DisconnectNamedPipe failed with error Xd\n",
           GetLastErrorO),
    return,
}

CloseHandle(PipeHandle),

```

**Формирование пустого списка DACL (Null DACL)**

При использовании API-функций для создания таких объектов, как файлы и именованные каналы, Windows NT и Windows 2000 позволяют приложениям задавать права доступа с помощью структуры *SECURITY\_ATTRIBUTES*

```

typedef struct _SECURITY_ATTRIBUTES {
    DWORD    nLength,
    LPVOID   lpSecurityDescriptor,
    BOOL     bInheritHandle
} SECURITY_ATTRIBUTES,

```

Поле *lpSecurityDescriptor* — указатель на структуру *SECURITY\_DESCRIPTOR*, определяющую права доступа к объекту Структура *SECURITY\_DESCRIPTOR* содержит поле DACL со списком пользователей и групп, имеющих доступ к объекту Если это поле равно *NULL*, объект доступен любому пользователю

Приложения не могут напрямую обращаться к структуре *SECURITY\_DESCRIPTOR* для этого существуют специальные API-функции. Например, чтобы создать пустой список DACL в структуре *SECURITY\_DESCRIPTOR*, необходимо сделать следующее:

- 1 Создайте и инициализируйте структуру *SECURITY\_DESCRIPTOR* с помощью API-функции *InitializeSecurityDescriptor*
- 2 Присвойте полю DACL значение *NULL* с помощью функции *SetSecurityDescriptorDacl*

Созданную структуру *SECURITY\_DESCRIPTOR* необходимо поместить в структуру *SECURITY\_ATTRIBUTES*, после чего ее можно использовать при вызове таких API-функций, как *CreateNamedPipe*

**И** Создание объектов структур *SECURITY\_ATTRIBUTES* и *SECURITY\_DESCRIPTOR*

```
SECURITY_ATTRIBUTES sa,
SECURITY_DESCRIPTOR sd,
```

```
// Инициализация нового объекта SECURITY_DESCRIPTOR для очистки значений
if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION)
    == 0)
{
    printf( InitializeSecurityDescriptor failed with error %d\n ,
           GetLastError0),
    return,
```

```
// Присвоение полю DACL структуры SECURITY_DESCRIPTOR значения NULL
if (SetSecurityDescriptorDacl(&sd TRUE, NULL, FALSE) == 0)
{
    printf( SetSecurityDescriptorDacl failed with error %d\n',
           GetLastError0),
    return,
```

```
// Назначение нового объекта SECURITY_DESCRIPTOR структуре SECURITY_ATTRIBUTES
sa nLength = sizeof(SECURITY_ATTRIBUTES),
sa lpSecurityDescriptor = &sd,
sa bInheritHandle = TRUE,
```

## Усовершенствованный сервер каналов

В листинге 4-1 приведен код сервера, обслуживающего только один экземпляр именованного канала. При этом все API-вызовы выполняются синхронно, то есть каждый вызов ждет окончания предыдущего запроса ввода-вывода. Чтобы позволить двум и более клиентам установить соединение, сервер должен поддерживать несколько экземпляров именованного канала, максимальное число которых ограничено значением параметра *nMaxInstances*. Функции *CreateNamedPipe*

Несколько экземпляров канала можно создать с помощью потоков, либо механизмов асинхронного ввода-вывода Win32, например перекрытого вво-

да-вывода и портов завершения. Механизмы асинхронного ввода-вывода позволяют одновременно обслуживать все экземпляры канала из одного потока приложения. Рассмотрим, как создать более сложный сервер с помощью потоков и перекрытого ввода вывода (подробнее о портах завершения — в главе 8).

## Потоки

Разработать сложный сервер, способный поддерживать более одного соединения с помощью потоков, не так уж трудно. Для каждого соединения необходимо создать отдельный поток и работать с ним, как с простым сервером. В листинге 4-2 показан код сервера, обслуживающего пять экземпляров именованного канала. Это приложение — эхо-сервер, получающий данные от клиента и отправляющий их обратно.

### Листинг 4-2. Использование потоков при реализации сложного сервера именованного канала

```
// Threads.cpp

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

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID IpParameter);

void main(void)

{
    HANDLE ThreadHandle;
    INT i;
    DWORD ThreadId;

    for(i = 0; i < NUM_PIPES;

        // Создание потока для обслуживания каждого экземпляра именованного канала
        if ((ThreadHandle = CreateThread(NULL, 0, PipeInstanceProc,
            NULL, 0, &ThreadId)) == NULL)
        {
            printfC'CreateThread failed with error X\n",
                GetLastError());
            return;
        }
        CloseHandle(ThreadHandle);

    printf("Press a key to stop the server\n");
    _getch();
}
```

Листинг 4-2. *(продолжение)*

```

II функция: PipeInstanceProc

// описание:
//      Эта функция обрабатывает один экземпляр именованного канала

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter)

    HANDLE PipeHandle;
    DWORD BytesRead;
    DWORD BytesWritten;
    CHAR Buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\A\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        NUM_PIPE_S, 0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)

        printf("CreateNamedPipe failed with error Xd\\n",
            GetLastErrorO);
        return 0;

// Обслуживание соединений клиентов в бесконечном цикле
while(1)
{
    if (ConnectNamedPipe(PipeHandle, NULL) == 0)
    {
        printf("ConnectNamedPipe failed with error Xd\\n",
            GetLastErrorO);

        <<* break;
        *..

        II Чтение данных и отправка их обратно клиенту до тех пор,
        // пока он не прекратит передачу
        while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
            &BytesRead, NULL) > 0)
        {
            printf("Echo Xd bytes to client\\n", BytesRead); <.u

            *. if (WriteFile(PipeHandle, Buffer, BytesRead,
                &BytesWritten, NULL) == 0) 3

                printf("WriteFile failed with error Xd\\n",
                    GetLastErrorO);
                break;
            }
        }
    }
}

```

**Листинг 4-2.** *{продолжение}*

```

if (DisconnectNamedPipe(PipeHandle) == 0)
{
    printfC'DisconnectNamedPipe failed with error %d\n",
        GetLastErrorO),
    break;

CloseHandle(PipeHandle);
return 0;

```

С помощью API-функции *CreateThread* сервер запускает пять потоков, каждый из которых выполняет функцию *PipeInstanceProc*. Функция *PipeInstanceProc* работает точно так же, как простой сервер (листинг 4-1). Сеанс связи завершается вызовом функции *DisconnectNamedPipe*, после чего приложение снова вызывает функцию *ConnectNamedPipe* с тем же описателем, чтобы обслужить другого клиента.

**Перекрытый ввод-вывод**

Это механизм асинхронного выполнения таких API-функций, как *ReadFile* и *WriteFile*. Прежде всего в функцию необходимо передать структуру *OVERLAPPED*, которая впоследствии будет использована для получения результатов запроса ввода-вывода с помощью API-функции *GetOverlappedResult*. Если функция вызывается с *OVERLAPPED*, результат возвращается немедленно.

Чтобы сервер обслуживал несколько экземпляров именованного канала с помощью перекрытого ввода-вывода, значение параметра *nMaxInstances* функции *CreateNamedPipe* должно быть больше 1, а параметр *dwOpenMode* — равен *FILE\_FLAG\_OVERLAPPED*. На листинге 4-3 показан код сервера, обслуживающего пять экземпляров именованного канала. Это приложение — эхо-сервер, который получает данные от клиента и отправляет их обратно.

Листинг 4-3. Использование перекрытого ввода-вывода при реализации усовершенствованного сервера именованных каналов

```

// Overlap.cpp

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

#define NUM_PIPES 5                                (C
#define BUFFER_SIZE 256

void main(void)                                    jefl
{
    if (i==1,i

    HANDLE PipeHandles[NUM_PIPES];
    DWORD BytesTransferred,
    \ CHAR Buffer[NUM_PIPES][BUFFER_SIZE];

```

## Листинг 4-3. (продолжение)

```

INT 1,
OVERLAPPED Ovlap[NUM_PIPES]; •
HANDLE Event[NUM_PIPES],

// Сервер должен хранить текущее состояние каждого канала Для этого
// предназначен массив DataRead Исходя из текущего состояния канала,
// сервер будет определять, какую операцию ввода-вывода необходимо выполнить.
BOOL DataRead[NUM_PIPES],

DWORD Ret;
DWORD Pipe;

for(i = 0, l < NUM_PIPES, i++)
{
    // Создание экземпляра именованного канала
    if ((PipeHandles[i] = CreateNamedPipe("\\\\.\\PIPE\\jiin",
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe for pipe Xd failed "
            "with error Xd\n", l, GetLastErrorO),
        return,

        // Для каждого экземпляра канала создается обработчик события,
        // который будет использоваться для мониторинга активности
        // операций перекрытого ввода-вывода
        if ((Eventfi] = CreateEvent(NULL, TRUE, FALSE, NULL))
            == NULL)
        {
            printf("CreateEvent for pipe Xd failed with error Xd\n",
                l, GetLastErrorO);
            continue,

            // Инициализация флага состояния, определяющего следует ли записывать
            // и читать из канала
            DataRead[i] = FALSE;

            ZeroMemory(&Ovlap[l], sizeof(OVERLAPPED));
            Ovlap[i].hEvent = Event[i];

            // Прослушивание клиентских соединений с помощью функции ConnectNamedPipeO
            if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
            {
                if (GetLastErrorO != ERROR_IO_PENDING)
                {
                    "••iw

```

ш. след. стр.



Листинг 4-3. *(продолжение)*

```

        printf("ConnectNamedPipe for pipe Xd failed with"
               " error Xd\n", l, GetLastError());
        CloseHandle(PipeHandles[i]);
        return,
    }
}

printf("Server is now running\n");

// Чтение и отправка данных обратно клиенту в бесконечном цикле
while(1)
{
    if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
        FALSE, INFINITE)) == WAIT_FAILED)
    {
        printf("WaitForMultipleObjects failed with error Xd\n",
            GetLastError());
        return;

        Pipe = Ret - WAIT_OBJECT_0,

        ResetEvent(Event[Pipe]);

        // Проверка результатов. При возникновении ошибки соединение
        // устанавливается заново, иначе выполняется чтение и запись

        if (GetOverlappedResult(PipeHandles[Pipe], &Ovlap[Pipe],
            &BytesTransferred, TRUE) == 0)
        {
            printf("GetOverlapped result failed Xd start over\n",
                GetLastError());

            if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
            {
                printf("DisconnectNamedPipe failed with error Xd\n",
                    GetLastError());
                return;
            }

            if (ConnectNamedPipe(PipeHandles[Pipe],
                &Ovlap[Pipe]) == 0)
            {
                if (GetLastError() != ERROR_IO_PENDING)
                {
                    // Обработка ошибки канала и закрытие описателя
                    printf("ConnectNamedPipe for pipe Xd failed with"

```

Листинг 4-3. (продолжение)

```

        " error Xd\n", i, GetLastError()),
        CloseHandle(PipeHandles[Pipe]),

    DataRead[Pipe] = FALSE;
}
else
{
    // Проверка состояния канала. Если значение переменной
    // DataRead равно FALSE, асинхронно читаем отправленные клиентом данные,
    // иначе отправляем данные обратно клиенту.

    if (DataRead[Pipe] == FALSE)

        // Подготовка к чтению данных от клиента путем
        // асинхронного вызова функции ReadFile

        ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
        Ovlap[Pipe].hEvent = Event[Pipe];

        if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
            BUFFER.SIZE, NULL, &Ovlap[Pipe]) == 0)

            if (GetLastErrorO != ERROR_IO_PENDING)

                printfC'ReadFile failed with error Xd\n",
                    GetLastErrorO);

        DataRead[Pipe] = TRUE;
    }
    else
    {
        // Отправка данных обратно клиенту путем
        // асинхронного вызова функции WriteFile
        printf("Received Xd bytes, echo bytes back\n",
            BytesTransferred);

        ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
        Ovlap[Pipe].hEvent = Event[Pipe];

        if (WriteFile(PipeHandles[Pipe], Buffer[Pipe],
            BytesTransferred, NULL, &Ovlap[Pipe]) == 0)
        {
            if (GetLastErrorO != ERROR_IO_PENDING)

```

см. след. стр.

Листинг 4-3. (продолжение)

```
pnntf( WriteFile failed with error %d\n
GetLastErrorO),
```

```
DataRead[Pipe] = FALSE,
```

```
} 133 n.'UH<>
```

Чтобы получить описатели каждого канала, приложение пять раз вызывает функцию *CreateNamedPipe*. Затем сервер начинает прослушивать каждый канал. Для этого он пять раз асинхронно вызывает функцию *ConnectNamedPipe*, передавая ей структуру *OVERLAPPED*. При подключении клиента все операции ввода-вывода обрабатываются асинхронно. По завершении соединения с клиентами сервер вызывает функцию *DisconnectNamedPipe*, а затем повторно иницирует прослушивание каждого канала с помощью функции *ConnectNamedPipe*.

## Олицетворение

Именованные каналы позволяют воспользоваться встроенными возможностями защиты Windows NT и Windows 2000 для управления доступом клиентов. Windows NT и Windows 2000 поддерживают так называемое олицетворение, позволяющее серверу выполняться в контексте безопасности клиента. Обычно сервер именованного канала работает в контексте безопасности процесса, который его запустил. Например, если сервер именованного канала запущен пользователем, обладающим привилегиями администратора, этот пользователь получает доступ практически к любым ресурсам Windows NT или Windows 2000. Данная ситуация не безопасна, ведь структура *SECURITY\_DESCRIPTOR*, переданная в функцию *CreateNamedPipe*, открывает доступ к каналу любым пользователям.

Установив соединение с клиентом с помощью функции *ConnectNamedPipe*, сервер может заставить свой поток выполняться в контексте безопасности клиента, вызвав API-функцию *ImpersonateNamedPipeClient*.

```
BOOL ImpersonateNamedPipeClient(
HANDLE hNamedPipe
```

Параметр *hNamedPipe* — это описатель экземпляра именованного канала, возвращенный функцией *CreateNamedPipe*. При вызове функции *ImpersonateNamedPipeClient* операционная система меняет контекст безопасности сервера на контекст безопасности клиента. В результате при обращении к разным ресурсам (например, файлам) сервер будет использовать права доступа клиента. Это позволяет сохранить управление доступом к ресурсам, независимо от того, кто запустил серверное приложение.

При работе в контексте безопасности клиента поток сервера использует один из четырех основных уровней олицетворения: анонимный (Анопу-

mous), идентификация (Identification), олицетворение (Impersonation) и делегирование (Delegation) Уровни олицетворения определяют степень, до которой сервер вправе представлять клиента Завершив сеанс связи, сервер должен вызвать функцию *RevertToSelf*, чтобы вернуться к первоначальному контексту безопасности Эта функция не имеет параметров

```
BOOL RevertToSelf(VOID),
```

## Детали реализации клиента

Реализация клиента именованного канала подразумевает разработку приложения, которое может подключаться к серверу Клиенты не могут создавать экземпляры именованных каналов и устанавливают соединение с уже существующими на сервере Для создания простого клиента выполните следующие действия

- 1 Для проверки наличия свободного экземпляра канала вызовите API-функцию *WaitNamedPipe*
- 2 Для установления соединения используйте API-функцию *CreateFile*
- 3 Для отправки и получения данных используйте API-функции *WriteFile* и *ReadFile*
- 4 Для завершения соединения используйте API-функцию *CloseHandle*

Перед установлением соединения клиент должен проверить наличие свободного экземпляра именованного канала функцией *WaitNamedPipe*

```
BOOL WaitNamedPipe(
    LPCISIR lpNamedPipeName,
    DWORD nTimeout
),
```

Параметр *lpNamedPipeName* определяет название именованного канала в формате UNC, а параметр *nTimeout* — сколько времени клиент будет ждать свободного экземпляра канала

В случае успешного выполнения функции *WaitNamedPipe* откройте экземпляр именованного канала с помощью API-функции *CreateFile*

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
```

Параметр *lpFileName* — название открываемого канала в формате UNC а параметр *dwDesiredAccess* задает режим доступа и должен быть равен *GENERIC\_READ* при чтении, или *GENERIC\_WRITE* — при записи данных в канал Можно указать оба флага, объединив их с помощью операции OR Режим до-

стуга должен соответствовать направлению передачи (параметр *dwOpenMode*), заданному при создании канала. Например, если канал создан с флагом *PLPE\_ACCESSINBOUND*, клиенту указывает флаг *GENERIC\_WRITE*.

Параметр *dwShareMode* должен быть равен 0, поскольку в каждый момент времени только один клиент может получить доступ к экземпляру канала. Параметр *ipSecurityAttributes* — *NULL*, если вы не хотите, чтобы дочерний процесс наследовал дескриптор клиента. Этот параметр нельзя использовать для управления доступом, так как с помощью функции *CreateFile* невозможно создать экземпляры именованного канала. Параметр *dwCreationDisposition* следует определить как *OPEN\_EXISTING*, тогда функция *CreateFile* будет возвращать ошибку, если канала не существует.

Параметр *dwFlagsAndAttributes* обязательно должен включать флаг *FILE\_ATTRIBUTE\_NORMAL*. Кроме того, можно указать флаги *FILE\_FLAG\_WRITE\_THROUGH*, *FILE\_FLAG\_OVERLAPPED* и *SECURITY\_QUIOS\_PRESENT*, объединив их с первым операцией ИЛИ. Флаги *FILE\_FLAG\_WRITE\_THROUGH* и *FILE\_FLAG\_OVERLAPPED* предназначены для управления вводом-выводом (см. описание функции *CreateNamedPipe*). Флаг *SECURITY\_QUIOS\_PRESENT* определяет уровень олицетворения клиента на сервере именованного канала. Уровни олицетворения задают степень свободы действий серверного процесса от имени клиентского. Клиент указывает эту информацию при подключении к серверу. Если клиент выбирает флаг *SECURITY\_QUIOS\_PRESENT*, он должен указать один или несколько флагов, перечисленных в следующем списке.

**III SECURITYANONYMOUS** — задает уровень анонимности. Сервер не может получить информацию о клиенте и выполняться в контексте безопасности клиента.

- **SECURITYIDENTIFICATION** — задает уровень идентификации. Сервер может получить информацию о клиенте, например идентификаторы и привилегии защиты, но не выполняться в контексте безопасности клиента. Это полезно, когда серверу необходимо идентифицировать клиента, но не нужно играть его роль.

**II SECURITYIMPERSONATION** — задает уровень олицетворения. Сервер может получить информацию о клиенте, а контекст безопасности клиента распространяется только на локальную систему. Этот флаг позволяет серверу получить доступ к любым локальным ресурсам на сервере, как если бы он был клиентом. Сервер не может представлять клиента на удаленных системах.

- **SECURITYDELEGATION** — задает уровень делегирования. Сервер может получить информацию о клиенте и выполняться в контексте безопасности клиента на его локальной системе и удаленных системах.

**ПРИМЕЧАНИЕ** В Windows NT не реализовано делегирование безопасности, поэтому флаг *SECURITYDELEGATION* следует применять, только если сервер работает под управлением Windows 2000.

**m SECURITYCONTEXTTRACKING** — задает динамический режим наблюдения защиты. Если этот флаг не указан, то режим статический.

**III SECURITYEFFECTIVEONLY** — указывает, что только включенные аспекты контекста безопасности клиента доступны серверу. Если этот флаг не назначен, серверу доступны любые аспекты контекста безопасности клиента.

Последний параметр функции *CreateFile* — *hTemplateFile*, не применяется при работе с именованными каналами и должен быть равен *NULL*. После успешного выполнения функции *CreateFile*, клиент может отправлять и принимать данные с помощью функций *ReadFile* и *WriteFile*. Завершив передачу, клиент закрывает соединение функцией *CloseHandle*.

В листинге 4-4 приведен код простого клиента именованного канала. При успешном соединении клиент отправляет серверу сообщение «This is a test».

Листинг 4-4. Простой клиент именованного канала

// Client.cpp

```

<include <windows.h>
(include <stdio.h>

(define PIPE_NAME "\\\\.\\Pipe\\jinr

void main(void)
{
    HANDLE PipeHandle,
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, INFINITE) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;

        // Создание описателя файла именованного канала
        if ((PipeHandle = CreateFile(PIPE_NAME,
            GENERIC_READ | GENERIC_WRITE, 0,
            (INSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
        {
            printf("CreateFile failed with error %d\n", GetLastError());
            return;

            if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten,
                NULL) == 0)

```

см. след. стр.

**Листинг 4-4. (продолжение)**

```

{
    printf( WriteFile failed with error %d\n , GetLastErrorQ),
    CloseHandle(PipeHandle),
    return,

    pnnntf( Wrote %d bytes , BytesWritten);

    CloseHandle(PipeHandle),

```

## Другие API-вызовы

Теперь обсудим ряд дополнительных функций, облегчающих работу с именованными каналами. Прежде всего, рассмотрим функции *CallNamedPipe* и *TransactNamedPipe*, которые могут существенно упростить код приложения. Обе функции выполняют операции чтения и записи в одном вызове. Функция *CallNamedPipe* позволяет клиенту подключиться к именованному каналу, работающему в режиме сообщений (и подождать, пока не освободится экземпляр канала), записать и считать данные, а затем закрыть канал. Фактически, она является полноценным клиентом именованного канала.

```

BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesRead,
    DWORD nTimeout
),

```

Параметр *lpNamedPipeName* определяет название именованного канала в формате UNC. Параметры *lpInBuffer* и *nInBufferSize* — адрес и размер буфера, используемого приложением для записи данных. Аналогично, *lpOutBuffer* и *nOutBufferSize* определяют адрес и размер буфера чтения. Параметр *lpBytesRead* содержит количество байтов, считанных из канала. Параметр *nTimeout* определяет время ожидания освобождения канала в миллисекундах.

Функция *TransactNamedPipe* используется как в клиентских, так и в серверных приложениях. Она объединяет операции чтения и записи в одном API-вызове. Это оптимизирует сетевой трафик за счет сокращения числа транзакций, выполненных через перенаправитель MSNP. Функция *TransactNamedPipe* определена так:

```

BOOL TransactNamedPipe(
    HANDLE hNamedPipe,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,

```

```

DWORD   nOutBufferSize,
LPDWORD lpBytesRead,
LPOVERLAPPED lpOverlapped

```

),

Параметр *hNamedPipe* — это описатель именованного канала, возвращенный функцией *CreateNamedPipe* или *CreateFile*. Параметры *ipInBuffer* и *nInBufferSize* определяют адрес и размер буфера, используемого приложением для записи данных. Аналогично *ipOutBuffer* и *nOutBufferSize* обозначают адрес и размер буфера чтения. Параметр *lpBytesRead* содержит количество байтов, считанных из канала. Параметр *lpOverlapped* позволяет функции *TransactNamedPipe* выполняться асинхронно путем перекрытого ввода-вывода.

Следующая группа функций *GetNamedPipeHandleState*, *SetNamedPipeHandleState* и *GetNamedPipeInfo*, — обеспечивает более гибкое взаимодействие сервера и клиента во время выполнения. Например, с помощью этих функций можно сменить режим работы именованного канала с побайтового на режим сообщений. Функция *GetNamedPipeHandleState* возвращает информацию о канале, включая режим работы, количество экземпляров канала и информацию о состоянии буферов. Информация, возвращаемая этой функцией, может изменяться в процессе работы именованного канала. Функция определена так:

```

BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpState,
    LPDWORD lpCurlInstances,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout,
    LPTSTR lpUserName,
    DWORD nMaxUserNameSize
),

```

Параметр *hNamedPipe* — это описатель именованного канала, возвращенный функцией *CreateNamedPipe* или *CreateFile*. Параметр *lpState* содержит текущий режим работы канала и принимает значения *PIPE\_NOWAIT* или *PIPE\_READMODE\_MESSAGE*. Параметр *lpCurlInstances* содержит текущее число экземпляров канала, *lpMaxCollectionCount* — максимальное число байтов, которые будут накоплены на компьютере клиента перед передачей на сервер, *lpCollectDataTimeout* — максимальное время в миллисекундах, которое может пройти, до того как удаленный клиент передаст информацию по сети. Параметры *lpUserName* и *nMaxUserNameSize* определяют буфер, содержащий заканчивающуюся 0 строку с именем пользователя клиентского приложения.

Функция *SetNamedPipeHandleState* позволяет изменить характеристики канала, возвращенные функцией *GetNamedPipeHandleState*.

```

BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,

```



```
LPDWORD lpMaxCollectionCount,
LPDWORD lpCollectDataTimeout
```

Параметр *hNamedPipe* — это описатель именованного канала, возвращенный функцией *CreateNamedPipe* или *CreateFile*. Параметр *ipMode* задает режим работы именованного канала. Параметр *IpMaxCollectionCount* содержит максимальное число байтов, которые будут накоплены на компьютере клиента перед передачей на сервер. Параметр *lpCollectDataTimeout* определяет максимальное время в миллисекундах, которое может пройти, до того как удаленный клиент передаст информацию по сети.

Функция *GetNamedPipeInfo* возвращает размер буферов и максимальное количество экземпляров канала:

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe,
    LPDWORD lpFlags,
    LPDWORD lpOutBufferSize,
    LPDWORD lpInBufferSize,
    LPDWORD lpMaxInstances
```

Параметр *hNamedPipe* — это описатель именованного канала, возвращенный функцией *CreateNamedPipe* или *CreateFile*. Параметр *lpFlags* указывает вид и режим работы именованного канала и определяет, является ли он сервером или клиентом. Параметры *IpOutBufferSize* и *IpInBufferSize* содержат размер исходящего и входящего внутренних буферов, *ipMaxInstance* — максимальное количество экземпляров канала.

Последняя функция — *PeekNamedPipe*, позволяет просмотреть находящиеся в канале данные, не удаляя их из внутреннего буфера. Например, проверить наличие входящих данных и избежать блокировки функции *ReadFile*. Кроме того, эта функция полезна, если необходимо проверить данные перед получением: приложение может скорректировать размер своего буфера в зависимости от размера входящего сообщения. Функция *PeekNamedPipe* описана так:

```
BOOL PeekNamedPipe(
    HANDLE hNamedPipe,
    LPVOID lpBuffer,
    DWORD nBufferSize,
    LPDWORD lpBytesRead,
    LPDWORD lpTotalBytesAvail,
    LPDWORD lpBytesLeftThisMessage
```

Параметр *hNamedPipe* — это описатель именованного канала, возвращенный функцией *CreateNamedPipe* или *CreateFile*. Параметры *lpBuffer* и *nBufferSize* определяют адрес и размер принимающего буфера. Параметр *lpBytesRead* содержит количество байтов, считанных из канала в буфер *lpBuffer*, *lpTotalBytesAvail* — общее количество байтов, которое можно считать из ка-

нала; *ipBytesLeftThisMessage* — оставшееся количество байт в сообщении, если канал работает в режиме сообщений. Если сообщение не помещается в буфер *ipBuffer*, данный параметр содержит оставшееся количество байт. Если именованный канал работает в побайтовом режиме, параметр всегда содержит 0.

## Платформа и производительность

В базе знаний Microsoft, к которой можно обратиться по адресу <http://support.microsoft.com/support/search>, описаны следующие проблемы и ограничения.

**Н Q100291 — ограничения на названия именованных каналов.** Канал `\\.\Pipe\MyPipes\Pipe1` невозможно создать при наличии канала `\\.\Pipe\MyPipes`. Название существующего канала нельзя использовать в качестве пути к другому каналу.

**И Q119218 — в именованный канал можно записать только 64 кб данных.** Если именованный канал работает в режиме сообщений, то при попытке записать данные, используя буфер большего размера, функция *WriteFile* вернет значение *FALSE*, а функция *GetLastError* — ошибку *ERROR\_MORE\_DATA*.

**III Q110148 — функции *WriteFile* или *ReadFile* возвращают ошибку *ERROR\_INVALID\_PARAMETER*,** если при работе с именованным каналом используется перекрытый ввод-вывод. Возможная причина такой ошибки — поля *Offset* и *OffsetHigh* структуры *OVERLAPPED* не равны 0.

- **Q180222 — функция *WaitNamedPipe* и ошибка с кодом 253 в Windows 95-** Если в Windows 95 в качестве первого параметра функции *WaitNamedPipe* передать неверное название канала, то функция *GetLastError* вернет ошибку с кодом 253, которой нет в списке возможных ошибок данной функции. Если то же самое проделать в Windows NT 4, код ошибки изменится на 161 (*ERROR\_BAD\_PATHNAME*). Обработайте ошибку 253 также, как 161.
- **Q141709 — максимум 49 соединений с одной рабочей станцией.** Если сервер именованных каналов создает 49 каналов, клиент на удаленном компьютере не может соединиться со следующим (50-м и далее) экземпляром именованного канала на этом сервере.
- **Q126645 — ошибка Access Denied (доступ запрещен) при открытии именованного канала из службы.** Если служба, запущенная под учетной записью Local System, попытается открыть именованный канал на компьютере с Windows NT, будет выдана ошибка Access Denied (с кодом 5).

## Резюме

В этой главе мы рассмотрели технологию именованных каналов, которые доставляют простую архитектуру клиент-сервер для надежной передачи

данных. Для передачи данных по сети данный интерфейс использует перенаправитель Windows. Основное преимущество именованных каналов — они позволяют воспользоваться встроенными возможностями защиты Windows NT и Windows 2000.

Эта глава завершает первую часть книги, в которой мы обсудили вопросы передачи данных средствами перенаправителя Windows. Во второй части мы рассмотрим технологию Winsock, которая позволяет обмениваться данными напрямую по транспортному протоколу.

# ИНТЕРФЕЙС ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ WINSOCK

Вторая часть книги посвящена программированию средствами Winsock на платформах Win32. Winsock — это сетевой интерфейс прикладного программирования, а не протокол, основной интерфейс доступа к разным базовым сетевым протоколам, реализованный на всех платформах Win32. Интерфейс Winsock унаследовал многое от реализации Berkeley (BSD) Sockets на платформах UNIX, работающих с множеством сетевых протоколов. В средах Win32 он стал абсолютно независимым от протокола, особенно с выпуском версии Winsock 2.

В следующих трех главах описаны основные характеристики протоколов и интерфейса Winsock, включая адресацию для каждого протокола и пример простого клиента и сервера Winsock. Мы рассмотрим новые технологические возможности интерфейса Winsock 2: поставщики службы транспорта, поставщики пространства имен и качество обслуживания (Quality of Service, V<sup>o</sup>S). В описании этих технологий есть некое несоответствие: хотя они и включены в спецификацию Winsock 2 и этот интерфейс поддерживается на всех современных платформах Win32 (за исключением ОС Windows CE), не все указанные в документации возможности реализованы на каждой из платформ. Об этих ограничениях мы упомянем дополнительно. Для изучения той части книги вы должны обладать базовыми знаниями о Winsock (или этах BSD) и знать клиент-серверную терминологию Winsock.

## Сетевые протоколы

Основная цель разработки спецификации Winsock 2 — создать независимый от протокола транспортный интерфейс. К его неоспоримым преимуществам относится предоставление единого привычного интерфейса сетевого программирования для различных транспортов сети. Впрочем, знать характеристики сетевых протоколов вам все же не помешает. В этой главе описаны различные аспекты работы с конкретными протоколами, а также некоторые основные сетевые правила. Кроме того, мы расскажем, как программно получить у Winsock сведения о протоколе, опишем основные шаги создания сокета для конкретных протоколов.

### Характеристики протоколов

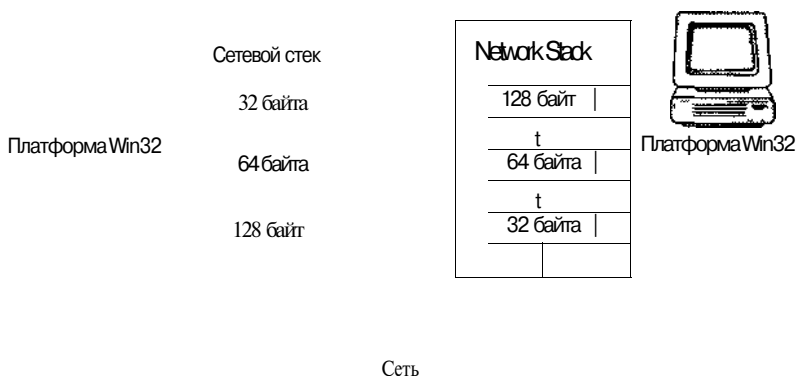
В этом разделе мы рассмотрим основные характеристики распространенных транспортных протоколов, а также то, как протокол функционирует в приложении.

#### Протокол, ориентированный на передачу сообщений

Протокол называют ориентированным на передачу сообщений, если для каждой команды записи он передает байты по сети в отдельном сообщении. Это также означает, что приемник получит данные в виде отдельного сообщения отправителя. Таким образом, приемник получит только одно сообщение. Например, на рис. 5-1 рабочая станция слева отправляет сообщения по 128, 64 и 32 байта рабочей станции справа. Принимающая рабочая станция дает три команды чтения с 256-байтным буфером. На каждый запрос последовательно возвращается 128, 64 и 32 байта. Первый запрос чтения не возвращает сразу три пакета, даже если все они уже получены. Таким образом, сохраняются границы сообщений, что часто необходимо для обмена структурированными данными. Например, в сетевой игре каждый участник отправляет другим игрокам пакет данных, с информацией о своей позиции. Программа, лежащая в основе такого обмена данными, очень проста: игрок запрашивает пакет данных и получает от другого участника игры именно один пакет данных с информацией о позиции другого игрока.

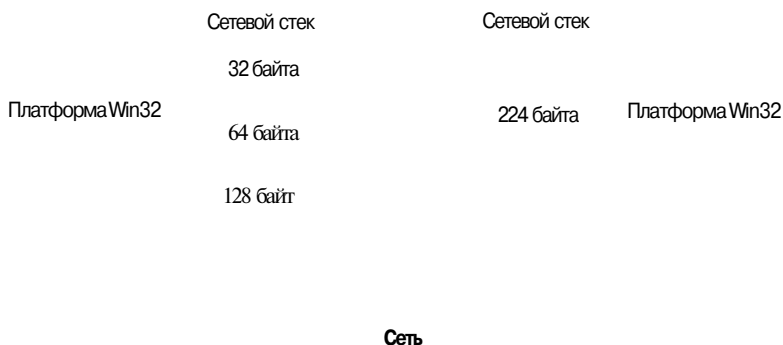
Протокол, не сохраняющий границы сообщений, обычно называют *протоколом, основанным на потоке*. Учтите, что термин «основанный на потоке» (stream-based) часто употребляют некорректно, подразумевая дополнительные характеристики. Потокковая служба непрерывно передает данные:

получатель считывает столько данных, сколько имеется в наличии, независимо от границ сообщений. Для отправителя это означает, что система может разбивать исходное сообщение на части или объединять несколько сообщений, чтобы сформировать большой пакет данных. На приемнике сетевой стек считывает данные по мере их поступления и буферизует для целевого процесса. Когда процесс запрашивает данные, система возвращает максимально возможное количество данных, не переполняющее буфер, предоставленный клиентским вызовом.



**Рис. 5-1. Службы дейтаграмм**

На рис. 5-2 отправитель передает пакеты данных по 128, 64 и 32 байт, однако стек локальной системы может принимать данные более крупными пакетами. В данном случае последние два пакета передаются вместе.



**Рис. 5-2. Поточковые службы**

Решение объединить дискретные пакеты данных зависит от нескольких факторов, например, от максимального размера блока передаваемой информации или применения алгоритма Nagle. В отношении TCP/IP применение Nagle заключается в том, что узел накапливает данные перед отправкой: ждет, пока накопится достаточно данных или истечет указанный тайм-аут. Партнер этого узла, перед тем как отправить узлу уведомление, ожидает исходящие данные в течение указанного времени. Это нужно, чтобы партнеру не

пришлось пересылать пакет данных «порожняком» — только с одним уведомлением. Отправка большого количества небольших по размеру пакетов данных неэффективна, поскольку влечет существенные издержки из-за многочисленных проверок на наличие ошибок и обмен подтверждениями. Со стороны получателя сетевой стек накапливает поступающие данные для конкретного процесса. Если получатель считывает данные, обладая 256-байтным буфером, то все 224 байта возвращаются сразу. Если приемник требует считать только 20 байт, система вернет только 20 байт.

## Псевдопоток

Система с ориентированным на передачу сообщений протоколом отправляет данные дискретными пакетами, которые получатель считывает и буферизует в пул так, чтобы получающее приложение считывало порции данных любого размера. Такую схему обмена данными зачастую и называют *псевдопоток*. Понять работу псевдопотока можно, скомбинировав отправителя на рис. 5-1 с получателем на рис. 5-2. Отправитель должен посылать каждый пакет данных по отдельности, но получатель может принимать их как угодно. В основном, перемещение данных псевдопоток можно рассматривать как обычный, основанный на потоке, протокол.

## Обмен данными, с соединением и без него

Любой протокол обычно предусматривает и ориентированные, и не ориентированные на соединение службы. Первые перед любым обменом данными устанавливают канал связи между двумя участвующими в обмене сторонами. Это гарантирует существование маршрута между двумя сторонами и то, что обе стороны будут корректно обмениваться информацией.

Впрочем, установление канала связи между двумя участниками влечет дополнительные издержки. Кроме того, большинство протоколов, ориентированных на передачу сообщений, гарантируют доставку данных, что еще больше увеличивает издержки, вызванные дополнительными вычислениями для проверки правильности передачи данных. С другой стороны, протокол, не ориентированный на передачу данных, не гарантирует, что приемник фактически принимает данные. Службы, не ориентированные на соединение, схожи с почтовой связью: отправитель адресует письмо определенному человеку и опускает его в почтовый ящик. Однако он не знает, существует ли вообще получатель письма, или не помешает ли сильная буря почтовой службе доставить послание.

## Надежность и порядок доставки сообщений

*Надежность и порядок доставки*, возможно, самые важные характеристики, которые следует учитывать при проектировании приложения для работы с определенным протоколом. В большинстве случаев надежность и порядок доставки неразрывно связаны с тем, ориентирован протокол на соединение или нет. Надежность, или гарантированная доставка, подразумевает, что каждый байт данных будет доставлен от отправителя указанному полу-

чательно без изменений. Ненадежный протокол не гарантирует ни доставку каждого байта, ни целостность данных.

Также необходимо принять во внимание порядок, в котором данные поступают получателю. Протокол, сохраняющий порядок данных, гарантирует что приемник получит эти данные в том порядке, в котором они были отправлены. Соответственно, протокол, не сохраняющий порядок байтов, не дает такой гарантии.

При установлении соединения стороны предпринимают дополнительные попытки по формированию свободного канала связи между собой, дабы гарантировать целостность данных и порядок доставки. В большинстве случаев протоколы, ориентированные на соединение, действительно гарантируют доставку.

Заметьте, что сохранение порядка пакетов не гарантирует автоматически целостность данных. Конечно, основное преимущество протоколов, не ориентированных на соединение, — это скорость: они не «заботятся» об установлении виртуального соединения с приемником. Зачем замедлять передачу данных проверкой на ошибки?

В общем, протоколы, не ориентированные на соединение, быстрее на порядок, чем протоколы, ориентированные на соединение, — проверки данных на целостность и уведомление об их успешном приеме намного усложняют отправку даже небольших порций данных. Так что дейтаграммы удобны для передачи не очень важных данных, например, для приложений, аналогичных уже приводившимся нами примеру с игрой: каждый игрок может использовать дейтаграммы, чтобы периодически отправлять информацию о своей позиции в игре всем другим игрокам по отдельности. Если один клиент пропускает пакет, то он быстро получает другой, что создает видимость непрерывной связи.

## Корректное завершение работы

*Корректное завершение работы* характерно только для протоколов, ориентированных на соединение. При этом одна сторона инициирует завершение сеанса связи, а другая — все еще имеет возможность считывать данные, задержавшиеся в канале связи или сетевом стеке. Ориентированный на соединение протокол, не поддерживающий корректного завершения работы, немедленно завершает сеанс связи, игнорируя любые данные, которые не были считаны приемником.

При использовании TCP каждая сторона соединения должна сначала выполнить все необходимые операции, чтобы окончательно завершить сеанс связи. Сторона — инициатор завершения сеанса, отправляет партнеру дейтаграмму с управляющим флагом FIN. Получив эту дейтаграмму, партнер отправляет управляющий флаг ACK стороне-инициатору, чтобы подтвердить получение флага FIN, но все еще может отправлять данные. Флаг FIN означает, что инициатор завершения сеанса отправлять данные больше не будет. Как только партнер завершит отправку своих данных, он также отправит **Ф** **а** **г** **FIN**, получение которого инициатор подтверждает флагом ACK. После Госса н с связи считается полностью завершенным.



## Широковещание данных

*Широковещание данных* подразумевает их передачу с одной рабочей станции всем остальным рабочим станциям ЛВС. Этой функцией обладают неориентированные на соединение протоколы, так как все компьютеры в ЛВС могут получать и обрабатывать широковещательные сообщения.

Недостаток широковещательных сообщений — их вынужден обрабатывать каждый компьютер. Например, пользователь передает сообщение всем станциям ЛВС, и сетевой адаптер каждого компьютера получает сообщение и помещает его в сетевой стек. Затем стек определяет, какие сетевые приложения должны получить это сообщение. Обычно большинству компьютеров в сети не нужны эти данные, и они их отбрасывают. Тем не менее, каждый вынужден тратить время на обработку пакета данных, чтобы проверить, нужны ли они для какого-нибудь приложения. Следствием является высокая рабочая нагрузка при широковещании, что может существенно замедлить работу в ЛВС. В общем, маршрутизаторы не транслируют широковещательных пакетов данных.

## Многоадресное вещание

Под *многоадресным, вещанием* понимается способность одного процесса передавать данные одному или более получателям. Методика присоединения процесса к многоадресному сеансу зависит от применяемого для передачи данных протокола.

Например, многоадресная передача по протоколу IP является видоизменной формой широковещания. Для нее необходимо, чтобы все заинтересованные в приеме и передаче узлы были членами особой группы. При присоединении процесса к группе многоадресного вещания на сетевом адаптере добавляется фильтр. Он заставляет сетевое оборудование обрабатывать и транслировать по сетевому стеку до соответствующего процесса только данные, предназначенные групповому адресу, к которому присоединился процесс. Многоадресная передача часто применяется в приложениях для видеоконференций. Подробнее о программировании многоадресного вещания средствами Winsock — в главе 11.

## Качество обслуживания

Управляя *качеством обслуживания* (Quality of Service, QoS), приложение может зарезервировать определенную часть пропускной способности сети для монопольного использования. Рассмотрим для примера воспроизведение видеопотока в реальном времени. Для его плавности и четкости видеоданные должны поступать из сети равномерно и с определенной скоростью. В недавнем прошлом плавное воспроизведение достигалось за счет накопления видеоданных в буфере. Если данные передавались неравномерно, паузы сглаживались кадрами из буфера. QoS позволяет резервировать определенную часть емкости канала связи, чтобы гарантировать равномерную передачу и считывание видеоданных. Теоретически это означает, что за счет использования QoS приложение может не буферизовать информацию. QoS посвящена глава 12.

## фрагментарные сообщения

*фрагментарные сообщения* (partial message) передают только ориентированные на сообщения протоколы. Предположим, приложению необходимо получить сообщение, а локальный компьютер принял лишь часть данных. Это обычное явление, особенно если компьютер-отправитель передает крупные сообщения. У локального компьютера может не хватить ресурсов, чтобы вместить сообщение целиком. На самом деле, большинство ориентированных на сообщения протоколов налагают разумные ограничения на максимально возможный размер дейтаграммы, чтобы такая ситуация не возникала часто.

Большинство дейтаграммных протоколов поддерживает передачу крупных сообщений, которые требуется переправлять по физической среде несколькими блоками. В результате, когда пользовательское приложение попытается прочесть сообщение, фактически будет принят лишь его фрагмент. Если протокол поддерживает фрагментарные сообщения, читатель уведомляется, что возвращаемые данные — лишь часть сообщения. Иначе базовый сетевой стек пытается сохранить фрагменты сообщения до тех пор, пока сообщение не поступит целиком. Если по какой-либо причине остатки сообщения не будут приняты, большинство ненадежных протоколов, не поддерживающих фрагментарные сообщения, просто отбросят неполную дейтаграмму.

## Маршрутизация

Важно учесть, является ли протокол *маршрутизируемым*. Если да, то между двумя рабочими станциями можно установить канал связи (виртуальный ориентированный на соединение либо канал передачи дейтаграмм), независимо от того, какая сетевая аппаратура их разделяет. Например, компьютер А находится в отдельной от компьютера В подсети. Между ними расположен маршрутизатор, соединяющий эти подсети. Маршрутизируемый протокол «знает», что эти компьютеры расположены в разных подсетях, поэтому направляет пакет данных маршрутизатору, который решает, как лучше переслать данные компьютеру В. Поскольку немаршрутизируемый протокол не способен передавать данные между сетями, маршрутизатор удаляет любые его пакеты. Маршрутизатор не пересылает пакет данных немаршрутизируемого протокола, даже если его адресат находится в подключенной подсети. Единственный немаршрутизируемый протокол, поддерживаемый платформами Win32 — NetBEUI.

## Другие характеристики

Каждый протокол, поддерживаемый на платформах Win32, обладает специфичными или уникальными характеристиками: например, порядком передачи байт или максимально допустимым размером пакета. Однако далеко не все эти характеристики важны для разработки Winsock-приложения. В Win-£. предусмотрен механизм перечисления каждого доступного поставщика протокола и опроса его характеристик (он описан разделе «Информация о протоколе» этой главы).

# Поддерживаемые протоколы

Весьма полезно, что платформы Win32 одновременно поддерживают несколько сетевых протоколов. Как уже упоминалось в главе 2, перенаправитель Windows гарантирует маршрутизацию сетевых запросов соответствующим протоколам и подсистемам. Впрочем, средствами Winsock вы можете написать сетевые приложения, напрямую использующие любой из этих протоколов. В главе 6 рассказывается, как происходит адресация компьютеров в сети при помощи протоколов, установленных на рабочей станции. Важно, что Winsock не зависит от протокола: большинство операций схожи при использовании любого протокола. Впрочем, способы адресации рабочих станций необходимо знать, чтобы определить местоположение и соединиться с сетевым партнером.

## Сетевые протоколы, поддерживаемые Win32

Платформы Win32 поддерживают разнообразные протоколы. Каждый протокол обычно способен работать в нескольких режимах. Например, протокол IP поддерживает как ориентированные на соединение поточные службы, так и службы дейтаграмм, не ориентированные на соединение. В табл. 5-1 перечислены основные доступные протоколы и некоторые поддерживаемые ими режимы работы.

**Табл. 5-1. Характеристики доступных протоколов**

Протокол	Имя	Тип сообщения	Установка соединения	Надежность	Порядок пакетов	Корректное завершение сеанса	Поддержка широко- вещания	Под-держка многоад-ресное™	QoS	Макс. размер сообщения (байт)
IP	MSAFD TCP	Поток	Да	Да	Да	Да	Нет	Нет	Нет	Без ограничений
	MSAFD UDP	Сообщение	Нет	Нет	Нет	Нет	Да	Да	Нет	65467
	RSVP TCP	Поток	Да	Да	Да	Да	Нет	Нет	Да	Без ограничений
	RSVP UDP	Сообщение	Нет	Нет	Нет	Нет	Да	Да	Да	65467
IPX/SPX	MSAFD nwln-kipx [IPX]	Сообщение	Нет	Нет	Нет	Нет	Да	Да	Нет	576
	MSAFD nwln-kspk [SPX]	Сообщение	Да	Да	Да	Нет	Нет	Нет	Нет	Без ограничений
	MSAFD nwln-kspk [SPX] [псевдо-поток]	Сообщение	Да	Да	Да	Нет	Нет	Нет	Нет	Без ограничений

**Табл. 5-1.** (продолжение)

Про- токол	Имя	Тип сооб- щения	Уста- новка соеди- нения	Надеж- ность	Порядок пакетов	Коррект- ное завер- шение сеанса	Под- держка широко- вещания	Под- держка многоад- ресности	ОoS	Макс. размер сообщения (байт)
	MSAFD nwln- kspix [SPXII]	Сооб- щение	Да	Да	Да	Да	Нет	Нет	Нет	Без оі ра- ничений
	MSAFD nwln- kspix [SPXII] [псевдо- поток]	Сооб- щение	Да	Да	Да	Да	Нет	Нет	Нет	Без огра- ничений
Net- BIOS	Sequen- ml Рас- kets (последо- ватель- ность пакетов)	Сооб- щение	Да	Да	Да	Нет	Нет	Нет	Нет	64 кб (65535)
	Datag- rams (дейтаг- раммы)	Сооб- щение	Нет	Нет	Нет	Нет	Да <sup>1</sup> [SP25]	Нет	Нет	64 кб (65535)
Apple- Talk	MSAFD Apple- Talk [ADSP]	Сооб- щение	Да	Да	Да	Да	Нет	Нет	Нет	64 кб (65535)
	MSAFD Apple- Talk [ADSP] [псевдо- поток]	Сооб- щение	Да	Да	Да	Да	Нет	Нет	Нет	Без огра- ничений
j	MSAFD Apple- Talk [PAP]	Сооб- щение	Да	Да	Да	Да	Нет	Нет	Нет	4096
on	MSAFD Apple- Talk [RTMP]	Поток	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Без огра- ничений
	MSAFD Apple- Talk [ZIP]	Поток	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Без огра- ничений
ATM	MSAFD ATM AAL5	Поток	Да	Нет	Да	Нет	Нет	Да	Да	Без огра- ничений

NetBIOS поддерживает отправку дейтаграмм как уникальным, так и групповым клиентам, общее широковещание не поддерживается »3,

см. след. стр.

Табл. 5-1. {продолжение}

Протокол	Имя	Тип сообщения	Установка соединения	Надежность	Порядок пакетов	Корректное завершение сеанса	Поддержка широко-вещания	Поддержка многоадресное™	QoS	Макс. размер сообщения (байт)
	Native ATM (AAL5)	Сообщение	Да	Нет	Да	Нет	Нет	Да	Да	Без ограничений
Infra-red sockets	MSAFD So-Irda [IrDA]	Погок	Да	Да	Да	Да	Нет	Нет	Нет	Без ограничений

## Сетевые протоколы в Windows CE

В отличие от других платформ Win32, Windows CE поддерживает только TCP/IP. Кроме того, Windows CE поддерживает только Winsock 1.1, поэтому большинство новых возможностей Winsock 2, описанных в этом разделе, не применимы к данной платформе. Windows CE поддерживает NetBIOS поверх TCP/IP при помощи перенаправителя, но не позволяет обращаться к этому протоколу ни через «родной» API-интерфейс NetBIOS, ни через Winsock.

## Информация о протоколе

Winsock 2 позволяет узнать, какие протоколы и с какими характеристиками установлены на рабочей станции. Для каждого рабочего режима протокола существует соответствующая запись каталога в рамках системы. Например, после установки TCP/IP в каталог будут занесены две записи для IP: одна для протокола TCP (надежного, с установлением соединения), вторая — для протокола UDP (ненадежного, без установления соединения).

Узнать об установленных сетевых протоколах можно с помощью функции *WSAEnumProtocols*:

```
int WSAEnumProtocols (
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    LPDWORD lpdwBufferLength
```

Она заменила функцию *EnumProtocols* из Winsock 1.1, применяемую в Windows CE. Единственное отличие: *WSAEnumProtocols* возвращает массив структур *WSAPROTOCOL\_INFO*, а *EnumProtocols* — массив структур *PROTOCOL\_INFO*, содержащий меньше полей, чем структура *WSAPROTOCOL\_INFO* (хотя информация та же). Структура *WSAPROTOCOL\_INFO* определена так:

```
typedef struct _WSAPROTOCOL_INFO {
    DWORD          dwServiceFlags1;
    DWORD          dwServiceFlags2;
    DWORD          dwServiceFlags3;
    DWORD          dwServiceFlags4;    < ш,
```

```

DWORD        dwProviderFlags;
GUID          ProviderId;
DWORD        dwCatalogEntryId;
WSAPROTOCOLCHAIN ProtocolChain;
int           iVersion;
int           iAddressFamily;
int           iMaxSockAddr;
int           iMinSockAddr;
int           iSocketType;
int           iProtocol;
int           iProtocolMaxOffset;
int           INetworkByteOrder;
int           ISecurityScheme;
DWORD        dwMessageSize;
DWORD        dwProviderReserved;
WCHAR        szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;

```

### Инициализация Winsock

Перед вызовом любой функции Winsock необходимо загрузить правильную версию библиотеки Winsock. Функция инициализации Winsock — *WSAStartup*:

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
```

Первый параметр — версия библиотеки Winsock, которую необходимо загрузить. На современных платформах Win32 используется версия 2.2. Единственное исключение — Windows CE, поддерживающая только Winsock 1.1. Для загрузки версии Winsock 2.2 укажите значение 0x0202, либо макрос *MAKEWORD(2, 2)*. Верхний байт определяет дополнительный номер версии, нижний — основной.

Второй параметр — структура *WSADATA*, возвращаемая по завершении вызова. Она содержит информацию о версии Winsock, загруженной функцией *WSAStartup*:

```

typedef struct WSADATA {
    WORD        wVersion;
    WORD        wHighVersion;
    char        szDescription[WSADESCRIPTION_LEN + 1]
    char        szSystemStatus[WSASYS_STATUS_LEN + 1]
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *   lpVendorInfo;
} WSADATA, FAR * LPWSADATA;

```

Единственная полезная информация, возвращаемая в структуре *WSADATA* — поля *wVersion* и *wHighVersion*. Поля, относящиеся к максимальному количеству сокетов и максимальному размеру UDP, следует получать из записи каталога для конкретного протокола. Ранее мы

*см. след. стр.*

говорили об этом, когда приводили описание *WSAEnumProtocols* и возвращаемых этой функцией структур.

Индивидуальные поля структуры *WSADATA* таковы:

*III wVersion* — версия Winsock, которую предполагает использовать ВЫЗОВ;

- *wHighVersion* — высшая версия Winsock, поддерживаемая загруженной библиотекой (как правило, то же значение, что и *wVersion*)

*III szDescription* — текстовое описание загруженной библиотеки;

*III szSystemStatits* — текстовая строка с соответствующей информацией о состоянии или конфигурации;

- *iMaxSockets* — максимальное количество сокетов (пропустите это поле для Winsock 2 и более поздних версий);

*Ж iMaxUdpDg* — максимальный размер дейтаграммы UDP;

- *ipVendorInfo* — информация об изготовителе (пропустите это поле для Winsock 2 и более поздних версий).

По завершении работы с библиотекой Winsock вызовите функцию *WSACleanup* для выгрузки библиотеки и освобождения ресурсов:

```
int WSACleanup (void);
```

Для каждого вызова *WSAStartup* необходимо согласованно вызывать *WSACleanup*, так как каждый стартовый вызов увеличивает значение эталонного счетчика ссылок на загруженные Winsock DLL. Чтобы уменьшить значение счетчика, требуется равное количество вызовов *WSACleanup*.

Обратите внимание: Winsock 2 полностью совместим со всеми вызовами функций Winsock 1.1. Поэтому приложение, написанное для Winsock 1.1, будет работать и с библиотекой Winsock 2- функции Winsock 1.1 сопоставляются их эквивалентам в Winsock 2.

Проще всего задать *WSAEnumProtocols* с *ipProtocolBuffer*, равным *NULL*, и *IpdwBufferLength* — равным 0. При вызове с *WSAENOBUFFS* произойдет ошибка, однако *IpdwBufferLength* будет содержать размер буфера, достаточный для возвращения всей информации о протоколах. Вызвав функцию с правильным размером буфера, вы получите несколько структур *WSAPROTOCOLINFO*. Просмотрите структуры в цикле и найдите запись протокола с необходимыми атрибутами Программа Enum.c на прилагаемом компакт-диске перечисляет все установленные протоколы и распечатывает их характеристики.

Особенно часто в структуре *WSAPROTOCOLINFO* используется поле *dwServiceFlags1* — битовая маска разных атрибутов протокола. В следующем списке перечислены битовые флаги и действия, которые иницируются, если данный флаг задан.

- *XP1 CONNECTIONLESS* — протокол передает данные без установления соединения; если флаг не задан — с установлением соединения.
- \* *XP1 GUARANTEED DELIVERY* — протокол гарантирует доставку данных получателю.

- ж* **XPI\_GUARANTEED\_ORDER** — протокол гарантирует доставку данных в порядке их отправки без дублирования, хотя саму доставку не гарантирует.
- щ* **XPI\_MESSAGEORIENTED** — протокол обрабатывает границы сообщений.
- ш* **xpiPSEUDOSTREAM** — протокол передает сообщения, но границы сообщений игнорируются приемником.
- III XPI\_GRACEFUL\_CLOSE** — протокол поддерживает двухфазное завершение сеанса (каждая сторона уведомляется о намерении другой завершить сеанс связи). Если этот флаг не задан, сеанс разрывается без предупреждения.
- щ* **XPI\_EXPEDITED\_DATA** — протокол поддерживает обмен *срочными* (out-of-band) данными
- т* **XPI\_CONNECT\_DATA** — протокол поддерживает передачу данных с запросом соединения.
- **XPIDISCONNECTDATA** — протокол поддерживает передачу данных с запросом разъединения.
- III XPI\_SUPPORTBROADCAST** — протокол поддерживает механизм широковещания.
- **XPI\_SUPPORT\_MULTIPOINT** — протокол поддерживает механизм многоадресной передачи данных.
- III XPI\_MULTIPOINTCONTROLPLANE** — *плоскость управления* (control plane) маршрутизируется, если флаг не задан — этого не происходит.
- **XP1\_MULTIPOINT\_DATA\_PLANE** — плоскость данных маршрутизируется, если флаг не задан — этого не происходит.
- III XPIQoS\_SUPPORTED** — протокол поддерживает запросы QoS.
- \* **XP1\_UNI\_SEND** — протокол однонаправленный и обеспечивает лишь отправку данных.
- \* **XPIUNIRECV** — протокол однонаправленный и обеспечивает лишь прием данных.
- \* **XP1\_IFS\_HANDLES** — дескрипторы сокета, возвращенные поставщиком, являются описателями файловой системы IFS и могут быть использованы в таких API-функциях, как *ReadFile* и *WriteFile*.
- III XPI\_PARTIAL\_MESSAGE** — флаг **MSG\_PARTIAL** поддерживается функциями *WSASend* и *WSASendTo*.

Для проверки наличия определенного свойства выберите соответствующий флаг и логически сложите его с полем *dwServiceFlags* 1. Если результат сложения ненулевой, протокол обладает указанным свойством, иначе — нет.

большинство из приведенных в списке флагов подробно описаны в следующих главах. Поэтому сейчас рассмотрим другие, не менее важные поля: *iProtocol*, *iSocketType* и *iAddressFamily*.



Поле *iProtocol* определяет, к какому протоколу относится данная запись. Поле *iSocketType* важно, если протокол способен работать в разных режимах, например, с установлением поточного или дейтаграммного соединения. И наконец, поле *lAddressFamily* позволяет выяснить корректную структуру адресации, применяемую данным протоколом. Эти три поля очень важны при создании сокета для конкретного протокола.

## Сокеты Windows

Рассмотрим, как доступные протоколы используют средства Winsock. Как вы уже, вероятно, знаете, этот интерфейс основан на понятии сокета. Сокет — это описатель поставщика транспорта. В Win32 сокет отличается от описателя файла, а потому представлен отдельным типом — *SOCKET*. Сокет создается одной из двух функций:

```
SOCKET WSASocket (
    int af,
    int type,
    int protocol,
    LPWSAProtocolInfo lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
```

```
SOCKET socket (
    int af,
    int type,
    int protocol
```

Первый параметр — *af*, определяет семейство адресов протокола. Например, если вы хотите создать UDP- или TCP-сокеты, подставьте константу *AF\_INET*, чтобы сослаться на протокол IP. Второй параметр — *type*, это тип сокета для данного протокола. Он может принимать одно из следующих значений: *SOCKSTREAM*, *SOCK\_DGRAM*, *SOCK\_SEQPACKET*, *SOCK\_RAW* и *SOCK\_RDM*. Третий параметр — *protocol*, указывает конкретный транспорт, если для данного семейства адресов и типа сокета существует несколько записей. В табл. 5-2 перечислены значения, используемые в полях семейства адресов, типа сокета и протокола для данного сетевого транспорта.

**Табл. 5-2. Параметры сокеты**

Протокол	Семейство адресов	Тип сокета	Протокол	
Internet Protocol (IP)	<i>AF_INET</i>	TCP UDP Простые сокеты	<i>SOCKSTREAM</i> <i>SOCK_DGRAM</i> <i>SOCK_RAW</i>	<i>IPPROTO_IP</i> <i>IPPROTO_TCP</i> <i>IPPROTO_UDP</i>
IPX/SPX	<i>AF_NS</i>	MSAFD nwn- kpx [IPX]	<i>SOCK_DGRAM</i>	<i>IPPROTO_IPX</i>

Табл. 5-2. (продолжение)

Протокол	Семейство адресов	Тип сокета	Протокол	
	<i>AFJPX</i>	MSAFD nwlncspx [SPX]	<i>SOCK_SEQ-PACKET</i>	<i>NSPROTOJPX</i>
		MSAFD nwlncspx [SPX]	<i>SOCKJSTREAM</i>	<i>NSPROTO SPX</i>
		[псевдопоток]		
		MSAFD nwlncspx [SPXII]	<i>SOCK_SEQ-PACKET</i>	<i>NSPROTOJPIX</i>
		MSAFD nwlncspx [SPXII]	<i>SOCKJSTREAM</i>	<i>NSPROTOJPIX</i>
		[псевдопоток]		
NetBIOS	<i>AFJVETBIOS</i>	Последовательные пакеты	<i>SOCK_SEQ-PACKET</i>	Номер LANA
		Дейтаграммы	<i>SOCKDGRAM</i>	Номер LANA
AppleTalk	<i>AFAPPLE-TALK</i>	MSAFD AppleTalk [ADSP]	<i>SOCKJUDM</i>	<i>ATPROTOADSP</i>
		MSAFD AppleTalk [ADSP]	<i>SOCKJSTREAM</i>	<i>ATPROTOADSP</i>
		[псевдопоток]		
		MSAFD AppleTalk [PAP]	<i>SOCK_RDM</i>	<i>ATPROTOPAP</i>
		MSAFD AppleTalk [RTMP]	<i>SOCKJDGRAM</i>	<i>DDPPROTO RTMP</i>
		MSAFD AppleTalk [ZIP]	<i>SOCKJDGRAM</i>	<i>DDPPROTOZIP</i>
ATM	<i>AF_ATM</i>	MSAFDATM AAL5	<i>SOCKJtAW</i>	<i>ATMPROTO_AAL5</i>
		Native ATM (AAL5)	<i>SOCK_RAW</i>	<i>ATMPROTO_AAL5</i>
Infrared Sockets	<i>AFJRDA</i>	MSAFDIrda [IrDA]	<i>SOCKJSTREAM</i>	<i>IRDA PROTO SOCK_STREAM</i>

Начальные три параметра для создания сокета подразделены на три уровня. Первый и самый важный — семейство адресов. Он указывает используемый в настоящее время протокол и ограничивает применение второго и третьего параметров. Например, семейство адресов ATM (*AF\_ATM*) позволяет использовать только простые сокеты (*SOCKJZAW*). Аналогично, выбор семейства адресов и типа сокета ограничивает выбор протокола.

Впрочем, можно передать в параметре *protocol* значение 0. В этом случае система выбирает поставщика транспорта, исходя из других двух параметров — *ofn type*. Перечисляя записи каталога для протоколов, проверьте значение поля *dwProviderFlags* из структуры *WSAPROTOCOLINFO*. Если оно равно *PFL\_MATCHES\_PROTOCOL\_ZERO* — это стандартный транспорт, применяемый, если в параметре протокола *socket* или *WSASocket* передано значение 0. Перечислив все протоколы с помощью *WSAEnumProtocols*, передайте структуру *WSAPROTOCOLINFO* в функцию *WSASocket* как параметр *ipProtocolInfo*.

Затем укажите константу *FROMPROTOCOLINFO* во всех трех параметрах (*a/, type и protocol*) — для них будут использоваться значения из переданной структуры *WSAPROTOCOL\_INFO*. Так указывается определенная запись протокола.

Теперь рассмотрим два последних флага из *WSASocket*. Параметр группы всегда равен 0, так как ни одна из версий Winsock не поддерживает группы сокетов. В параметре *dwFlags* указывают один или несколько следующих флагов:

- *WSA\_FLAG\_OVERLAPPED*,
- WSA\_FLAG\_MULTIPOINT\_C\_ROOT*,
- WSA\_FLAG\_MULTIPOINT\_C\_LEAF*,
- WSA\_FLAG\_MULTIPOINT\_D\_ROOT*,
- WSA\_FLAG\_MULTIPOINT\_D\_LEAF*

Первый флаг — *WSA\_FLAG\_OVERLAPPED*, указывает, что данный сокет допускает перекрытый ввод-вывод — это один из механизмов связи, предусмотренных в Winsock (см. главу 8). Если вы создаете сокет функцией *socket*, флаг *WSA\_FLAG\_OVERLAPPED* задан по умолчанию. В общем, рекомендуется всегда задавать этот флаг при использовании *WSASocket*. Последние четыре флага относятся к сокетам многоадресного вещания.

## Простые сокет

При создании сокета функцией *WSASocket* вы можете передать вызову структуру *WSAPROTOCOL\_INFO*, чтобы определить тип сокета, который хотите создать. Впрочем, вы можете создавать типы сокетов, для которых нет записи в каталоге поставщиков транспорта. Лучшим пример тому — *простые сокет* (raw socket) под протоколом IP. Это одна из форм связи, позволяющая инкапсулировать другие протоколы, например Internet Control Message Protocol (ICMP) в пакеты UDP. Протокол ICMP доставляет управляющие и информационные сообщения, а также уведомления об ошибках между узлами в Интернете. Поскольку ICMP не предусматривает средств доставки данных, он работает не на том же уровне, что и протоколы UDP или TCP, а относится к уровню протокола IP. Подробнее о простых сокетах — в главе 13.

## Информация о платформах

Windows 95 изначально поддерживает спецификацию Winsock 1.1. Корпорация Microsoft предоставила возможность бесплатно загружать обновления для Winsock 2 по адресу <http://www.microsoft.com/windows95/downloads/>. Кроме того, доступен комплект разработчика — Winsock 2 SDK, содержащий заголовочные файлы и библиотеки, необходимые для компиляции приложений Winsock 2. Платформы Windows 98, Windows NT 4 и Windows 2000 изначально поддерживают Winsock 2. Windows CE поддерживает только Winsock 1.1.

Поставщики транспорта поддерживаются с рядом ограничений. Windows CE поддерживает только протоколы TCP/IP и Infrared Sockets (инфракрасные сокет). В Windows 95 и Windows 98 поставщики транспорта NetBIOS (транс-

порты с семейством адресов *AF NETBIOS*) недоступны из Winsock. При вызове *WSAEnumProtocols* ни один из поставщиков транспорта NetBIOS не будет перечислен, даже если установлен на компьютере. Впрочем, обратиться к NetBIOS можно через «родной» интерфейс NetBIOS (см главу 1). Упомянутые поставщики RSVP (предоставляет функции QoS) и ATM изначально поддерживаются в Windows 98 и Windows 2000.

## Winsock и модель OSI

Теперь обсудим, как некоторые из понятий, описанных в этой главе, относятся к модели OSI (рис. 1-1). Поставщики транспорта из каталога Winsock, перечисленные *WSAEnumProtocols*, работают на транспортном уровне модели OSI, то есть каждый из них обеспечивает обмен данными. Впрочем, все они относятся к какому-то протоколу, а сетевой протокол работает на сетевом уровне, поскольку обуславливает способ адресации каждого узла в сети. Например, UDP и TCP — это транспорты, хотя оба относятся к протоколу IP.

Интерфейс Winsock расположен между сеансовым и транспортным уровнями. Winsock позволяет открывать и закрывать сеанс связи и управлять им для любого данного транспорта. Под управлением Windows три верхних уровня прикладной, представительский и сеансовый, — в основном относятся к вашему приложению Winsock. Другими словами, приложение Winsock управляет всеми аспектами сеанса связи и при необходимости форматирует данные согласно целям программы.

## Выбор соответствующего протокола

При разработке сетевого приложения вы можете выбрать базовый протокол из числа имеющихся. Если приложению необходимо осуществлять связь по определенному протоколу, выбор не богат. Однако разрабатывая приложение «с нуля», лучше выбрать TCP/IP, поскольку этот протокол распространен и широко применяется в продуктах Microsoft, AppleTalk, NetBIOS и IPX/SPX. Microsoft поставляет для совместимости с другими операционными системами и существующими приложениями. Например, вместе с Windows 95 по умолчанию устанавливаются протоколы NetBEUI и IPX/SPX.

По мере роста популярности Интернета большинство организаций все шире используют TCP/IP. Это также основной протокол Windows 2000. В связи с этим NetBIOS будет применяться все реже. Учтите и активную поддержку Microsoft-реализации TCP/IP: ошибки в нем исправляются значительно чаще и быстрее, чем в других протоколах.

Таким образом, TCP/IP — это стратегический протокол для сетевых приложений. Помимо этого Microsoft активно поддерживает сети ATM. Если вы можете позволить себе разрабатывать приложение, функционирующее исключительно в сетях ATM, возьмите за основу ATM-функции из Wmsock. Пользователям TCP/IP следует иметь в виду, что сети ATM можно сконфигурировать для эмуляции TCP/IP, и этот механизм работает довольно хорошо, естественно,<sup>3</sup> Д<sup>с</sup> мы привели далеко не все факторы, учитываемые при разработке сетевого приложения.

## Резюме

В этой главе мы обсудили основные характеристики, которые обязательно нужно учитывать при выборе базового сетевого транспорта для проектируемого приложения. Мы рассмотрели, как программным путем получить список установленных в системе поставщиков транспорта и информацию об определенном свойстве протокола. И наконец, дали рекомендации как создать сокет для любого сетевого транспорта с помощью параметров функции *WSASocket* или *socket*, а также опрашивать запись каталога транспортов функцией *WSAEnumProtocols* и передавать структуру *WSAPROTOCOLINFO* функции *WSASocket*.

В следующей главе мы расскажем о способах адресации для всех широко распространенных протоколов.

## Семейства адресов и разрешение имен

Для связи средствами Winsock важны механизмы адресации рабочих станций в конкретных протоколах. В этой главе рассматриваются протоколы, поддерживаемые Winsock, а также порядок разрешения адресов разных семейств каждым протоколом. В Winsock 2 реализовано несколько новых независимых от протокола функций, которые можно использовать с сокетами любых семейств адресов. Впрочем, в большинстве случаев у каждого семейства свой механизм разрешения адресов с помощью функции либо через параметр, переданный функции *getsockopt*.

Материал этой главы содержит лишь основные понятия, рассказывает о формировании структуры адресов для каждого семейства протоколов. В главе 10 рассматриваются функции регистрации и разрешения имен, оповещающие о службе данного семейства протоколов (это не совсем то же, что разрешение имени). Там же вы найдете дополнительную информацию о различиях между прямым и обычным разрешением имен и оповещением о службе.

Для каждого семейства имен мы рассмотрим основы адресации компьютера в сети. Затем покажем, как создать сокет для каждого семейства. Кроме этого, будет описана специфика каждого протокола в методике разрешения имен.

### Протркол IP

Internet Protocol (IP) широко используется в Интернете, поддерживается большинством ОС и применяется как в *локальных* (local area networks, LAN), так и в *глобальных сетях* (wide area networks, WAN). IP не требует установления соединения и не гарантирует доставку данных. Поэтому для передачи данных поверх IP используются два протокола более высокого уровня: TCP и UDP.

### Протокол TCP

ransmission Control Protocol (TCP) реализует связь с установлением соединения, обеспечивает надежную безошибочную передачу данных между двумя компьютерами. Когда приложения связываются по TCP, осуществляется

Ртуальное соединение исходного компьютера с целевым, после чего между ними возможен двунаправленный обмен данными.

## Протокол UDP

Связь без установления соединения выполняется при помощи User Datagram Protocol (UDP). Не гарантируя надежности, UDP может осуществлять передачу данных множеству адресатов и принимать данные от множества источников. Например, данные, отправляемые клиентом на сервер, передаются немедленно, независимо от того, готов ли сервер к приему. При получении данных от клиента, сервер не подтверждает их прием. Данные передаются в виде дейтаграмм.

И TCP, и UDP передают данные по IP, поэтому обычно говорят об использовании TCP/IP или UDP/IP. В Winsock для IP-соединений предусмотрено семейство адресов *AF\_INET*, определенное в файлах Winsock.h и Winsock2.h.

## Адресация

При использовании IP компьютерам назначается IP-адрес, состоящий из 32 бит, официально называемый IP-адресом версии 4 (IPv4). Для взаимодействия с сервером по TCP или UDP клиент должен указать IP-адрес сервера и номер порта службы. Чтобы прослушивать входящие запросы клиента, сервер также должен указать IP-адрес и номер порта. В Winsock IP-адрес и порт службы задают в структуре *SOCKADDR\_IN*.

```
struct sockaddr_in
{
    short          sin_family;
    u_short        sin_port;
    struct in_addr sin_addr;
    char           sin_zero[8];
};
```

Поле *sin\_family* должно быть равно *AF\_INET*: этим Winsock сообщают об использовании семейства адресов IP.

### IP версии 6

IP версии 6 — обновленная спецификация, позволяющая использовать большее адресное пространство. Оно понадобится в недалеком будущем, когда возможности IP4 будут исчерпаны. Множество заголовков файлов Winsock содержит условное описание структур IPv6, однако ни одна из 32-битных платформ, включая Windows 2000, не обеспечивает работу сетевого стека IPv6. Исследователи из Microsoft Research представили экспериментальный стек IPv6 по адресу <http://research.microsoft.com/msripv6/>. Впрочем, он не поддерживается, а потому здесь мы не будем обсуждать особенности версии 6.

Поле *sin\_port* задает, какой коммуникационный порт TCP или UDP будет использован для идентификации службы сервера. Приложения должны быть очень внимательны при выборе порта, поскольку некоторые доступные порты зарезервированы для использования популярными службами: такими, как File Transfer Protocol (FTP) и Hypertext Transfer Protocol (HTTP). Эти порты

обслуживаются и распределяются центром Internet Assigned Numbers Authority (IANA), их описание см. в RFC 1700. По сути, номера портов делят на три категории: стандартные, зарегистрированные и динамические и (или) частные порты. Диапазоны портов:

- и* **0-Ю23** — управляются IANA и зарезервированы для стандартных служб;
- и* **1024-49151** — зарегистрированы IANA и могут использоваться процессами и программами;
- являются динамическими и (или) частными.

Во избежание накладок с портами, уже занятыми системой или другим приложением, ваша программа должна выбирать зарегистрированные порты в диапазоне 1024-49151. Порты 49152-65535 также можно задействовать свободно — с ними не связаны никакие стандартные службы. Если при использовании API-функции *bind* ваше приложение попытается выбрать порт, уже занятый другим приложением на узле, то система вернет ошибку *WSAEADDRINUSE*. Подробнее о процедуре привязки к порту в Winsock — в главе 7.

Поле *sin\_addr* структуры *SOCKADDR\_IN* хранит IP-адрес в 4-байтном виде с типом *unsigned long int*. В зависимости от того, как это поле использовано, оно может представлять и локальный, и удаленный IP-адрес. IP-адреса обычно задают в точечной нотации: a.b.c.d. Здесь каждая буква представляет число для каждого байта и задается слева направо (все четыре байта с типом *unsigned long int*). И наконец, поле *sin\_zero* играет роль простого заполнителя, чтобы структура *SOCKADDR\_IN* по размеру равнялась структуре *SOCKADDR*.

Полезная вспомогательная функция *inet\_addr* преобразует IP-адрес из точечной нотации в 32-битное длинное целое без знака:

```
unsigned long inet_addr(  
    const char FAR *cp  
);
```

Поле *cp* является строкой, заканчивающейся нулевым символом, здесь задается IP-адрес в точечной нотации. Заметьте, что эта функция в качестве результата возвращает IP-адрес, представленный 32-битным числом с сетевым порядком следования байт (*network-byte order*). Краткое описание этого порядка см. в разделе «Порядок байт».

## Специальные адреса

В некоторых ситуациях на поведение сокета влияют два специальных IP-адреса. Специальный адрес *INADDR\_ANY* позволяет серверному приложению слушать клиента через любой сетевой интерфейс на несущем компьютере.

обычно приложения сервера используют этот адрес, чтобы привязать сокет к локальному интерфейсу для прослушивания соединений. Если на компьютере несколько сетевых адаптеров, то этот адрес позволит отдельному приложению получать отклики от нескольких интерфейсов.

Второй специальный адрес — *INADDR\_BROADCAST*, позволяет широкове-Щательно рассылать UDP-дейтаграммы по IP-сети. Для его использования



необходимо в приложении задать параметр сокета *SO\_BROADCAST* (см также главу 9)

## Порядок байт

Разные процессоры в зависимости от конструкции представляют числа в одном из двух порядков байт *big-endian* или *little-endian*. Например, процессоры Intel x86 представляют многобайтные числа, следуя от менее значимого к более значимому байту (*little-endian*). Если номер порта и IP-адрес хранятся в памяти компьютера как многобайтные числа, они представляются в *системном порядке* (*host-byte-order*). Впрочем, когда IP-адрес или номер порта задаются по сети, стандарты Интернета требуют, чтобы многобайтные значения представлялись от старшего байта к младшему (в порядке *big-endian*), что обычно называется *сетевым порядком* (*network-byte order*).

Есть целый ряд функций для преобразования многобайтных чисел из системного порядка в сетевой и обратно. Например, четыре следующих API-функции преобразуют числа из системного порядка в сетевой

```
u_long htonl(u_long hostlong),
```

```
int WSAhtonl(
    SOCKET s,
    u_long hostlong,
    u_long FAR * lpnetlong
```

```
u_short htons(u_short hostshort);
```

```
int WSAhtons(
    SOCKET s,
    u_short hostshort,
    u_short FAR * lpnetshort
);
```

Параметры *hostlong* функций *htonl* и *WSAhtonl* — четырехбайтные числа с системным порядком следования байт. Функция *htonl* возвращает число с сетевым порядком, а *WSAhtonl* — число с сетевым порядком через параметр *lpnetlong*. Параметр *hostshort* функций *htons* и *WSAhtons* является двухбайтным числом с системным порядком. Функция *htons* возвращает число, как двухбайтное значение с сетевым порядком, тогда как функция *WSAhtons* возвращает такое число через параметр *lpnetshort*.

Следующие четыре функции решают обратную задачу: переставляют байты из сетевого порядка в системный

```
u_long ntohl(u_long netlong);
```

```
int WSANTohl(
    SOCKET s,
    u_long netlong,
    u_long FAR * lphostlong
```

```
u_short ntohs(u_short netshort);

mt WSANtohs(
    SOCKET s,
    u_short netshort,
    u_short FAR * lphostshort
);
```

А сейчас продемонстрируем, **как** создать структуру *SOCKADDR\_IN* при помощи уже описанных функций *inet\_addr* и *htons*.

```
SOCKADDR_IN InternetAddr,
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET,

// Преобразование адреса 136.149.3.29 из десятично-точечной нотации в
// 4-байтное целое число и присвоение результата sm_addr

InternetAddr.sin_addr s_addr = inet_addr("136.149.3 29"),

// Переменная nPortId хранится в системном порядке. Преобразование
// nPortId к сетевому порядку и присвоение результата sm_port.

InternetAddr.sin_port = htons(nPortId);
```

Теперь подготовим сокет для соединения по TCP или UDP.

## Создание сокета

Создание IP-сокета позволит приложениям осуществлять подключение через TCP, UDP и протоколы IP. Для открытия IP-сокета при помощи протокола TCP, вызовите функцию *socket* или *WSASocket* с семейством адресов *AF\_INET* и типом сокета *SOCK\_STREAM*, а также присвойте значение 0 полю протокола

```
s = socket(AF_INET, SOCKSTREAM, 0),
s = WSASocket(AF_INET, SOCKSTREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED),
```

Чтобы открыть IP-сокет при помощи протокола UDP, вместо *SOCKSTREAM* укажите тип сокета *SOCK\_DGRAM* в функции *socket* или *WSASocket*. Также можно открывать сокет для связи непосредственно по IP — для этого задайте тип сокета *SOCK\_RAW*. Подробнее об этом параметре — в главе 13.

## Разрешение имен

Для подключения к узлу по IP Winsock-приложение должно знать IP-адрес этого узла, сложный для запоминания пользователем. Большинство людей более охотно обращаются к компьютерам при помощи легко запоминаемых имен узлов, а не IP-адресов. В Winsock предусмотрено две функции для разрешения имени в IP-адрес

Функции *gethostbyname* и *WSAAsyncGetHostByName* отыскивают в базе данных узла сведения об узле, соответствующие его имени. Обе функции возвращают структуру *HOSTENT*.

```
struct hostent
```

```
    char FAR *      h_name,
    char FAR * FAR * h_aliases,
    short           h_addrtype,
    short           h_length,
    char FAR * FAR * h_addrj.ist,
```

Поле *hname* является официальным именем узла. Если в сети используется *доменная система имен* (Domain Name System, DNS), в качестве имени сервера будет возвращено полное имя домена (Fully Qualified Domain Name, FQDN). Если в сети применяется локальный файл узлов (*hosts*, *lmhosts*) — это первая запись после IP-адреса. Поле *h\_aliases* — массив, завершающийся нулем (null-terminated array) дополнительных имен узла. Поле *h\_addrtype* представляет возвращаемое семейство адресов. Поле *h\_length* описывает длину в байтах каждого адреса из поля *h\_addrj.ist*. Поле *h\_addrj.ist* — массив, завершающийся 0 и содержащий IP-адреса узла (Узел может иметь несколько IP-адресов). Каждый адрес в этом массиве представлен в сетевом порядке. Обычно приложение использует первый адрес из массива. Впрочем, при получении нескольких адресов, приложение должно выбирать адрес случайным образом из числа доступных, а не упорно использовать первый.

API-функция *gethostbyname* определена так:

```
struct hostent FAR * gethostbyname (
    const char FAR * name
```

Параметр *name* представляет дружественное имя искомого узла. При успешном выполнении функции возвращается указатель на структуру *HOSTENT*, которая хранится в системной памяти. Приложение не должно полагать, что эти сведения непременно статичны. Поскольку эта память обслуживается системой, оно не должно освобождать возвращенную структуру.

*WSAAsyncGetHostByName* — асинхронная версия функции *gethostbyname*, оповещающая приложение о завершении своего выполнения с помощью сообщений. Windows

```
HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR * buf,
    int buflen
```

Параметр *hWnd* — дескриптор окна, которое получит сообщение по завершении выполнения асинхронного запроса. Параметр *wMsg* — Windows-

сообщение, которое будет возвращено по завершении выполнения асинхронного запроса. Параметр *name*— дружественное имя искомого узла. Параметр *buf*— указатель на область данных, куда помещается *HOSTENT*. Этот буфер должен быть больше структуры *HOSTENT* и иметь размер, определенный *BMAXGETHOSTSTRUCT*.

Стоит упомянуть еще две функции поиска сведений об узле *gethostbyaddr* и *WSAAsyncGetHostByAddr*. Они полезны, когда вы знаете IP-адрес узла и хотите найти его понятное пользователю имя. Функция *gethostbyaddr* определена так

```
struct HOSTENT FAR * gethostbyaddr(
    const char FAR * addr,
    int len,
    int type
),
```

Параметр *addr* — указатель на IP-адрес в сетевом порядке. Параметр *len* задает длину параметра *addr* в байтах. Параметр *type* должен иметь значение *AF\_INET* (IP-адрес). *WSAAsyncGetHostByAddr* — асинхронная версия функции *gethostbyaddr*.

## Номера портов

Помимо IP-адреса удаленного компьютера для подключения к службе на локальном или удаленном компьютере приложение должно знать номер порта службы. При использовании TCP и UDP приложение решает, через какой порт связаться. Существуют стандартные номера портов, зарезервированные для служб сервера, поддерживающих протоколы более высокого уровня, чем TCP. Например, порт 21 зарезервирован для FTP, 80 — для HTTP. Как уже упоминалось, стандартные службы обычно используют порты 1-1023. Поэтому если вы разрабатываете TCP-приложение, которое не использует ни одну из известных служб, старайтесь задействовать порты с номером больше 1023. Вы можете узнать номера портов, используемых стандартными службами, вызвав функцию *getservbyname* или *WSAAsyncGetServByName*. Эти функции просто извлекают статическую информацию из файла с именем *services*. В Windows 95 и Windows 98 файл служб расположен в папке %WINDOWS%, а в Windows NT и Windows 2000 - в %WINDOWS%\System32\Drivers\Etc. Функция *getservbyname* определена так

```
struct servant FAR * getservbyname(
    const char FAR * name,
    const char FAR * proto
)
```

Параметр *name* представляет имя искомой службы. Например, если вы пытаетесь найти порт, используемый FTP, присвойте параметру *name* указатель на строку «ftp». Параметр *proto* иногда ссылается на строку, указывающую протокол, под которым зарегистрирована служба из параметра *name*. Функция *WSAAsyncGetServByName* — асинхронная версия *getservbyname*.

В Windows 2000 применен новый динамический метод регистрации и запроса информации о службах для TCP и UDP. Серверные приложения могут за-

регистрировать имя службы, IP-адрес и номер порта службы при помощи функции *WSASetService*. Клиентские приложения запрашивают информацию об этих службах при помощи комбинации API-функций *WSALookupServiceBegin*, *WSALookupServiceNext* и *WSALookupServiceEnd* (см также главу 10)

## Инфракрасные сокеты

*Сокеты инфракрасного канала* (Infrared sockets, IrSock) — новая интересная технология, впервые реализованная в Windows CE. Они позволяют подключаться двум компьютерам по инфракрасному последовательному порту. Инфракрасная связь теперь доступна в Windows 98 и в Windows 2000. Инфракрасные сокеты отличаются от традиционных тем, что учитывают непостоянство доступности переносных устройств. В этой технологии применена новая модель разрешения имен.

### Адресация

Поскольку большинство компьютеров с устройствами инфракрасной связи (Infrared Data Association, IrDA) мобильны, традиционные схемы разрешения имен не эффективны. Обычные статические ресурсы, такие как серверы имен, бесполезны, когда сетевой клиент перемещается по сети и за ее пределы. Для решения этой проблемы IrSock ищет ресурсы в радиусе связи произвольным образом, не налагая нагрузки на всю сеть и не применяя стандартные функции службы имен Winsock или IP-адреса. Вместо этого служба имен встроена в поток связи, а для поддержки служб, относящихся к последовательным инфракрасным портам, введено новое семейство адресов. Структура адреса IrSock содержит имя службы с описанием приложения, используемого в вызовах для привязки и подключения, а также *идентификатор устройства* (device identifier), определяющий устройство, на котором выполняется данная служба. Эта пара аналогична IP-адресу и номеру порта, используемым в обычных TCP/IP-сокетах. Структура адреса IrSock такова:

```
typedef struct sockaddr_irda {  
    u_short    IrdaAddressFamily,  
    u_char     IrdaDeviceID[4],  
    char       IrdaServiceName[25],  
> SOCKADDR_IRDA
```

Поле *IrdaAddressFamily* всегда равно *AF\_IRDA*. Параметр *IrdaDeviceID* — четырехсимвольная строка, уникально идентифицирующая устройство, на котором запущена определенная служба. Это поле игнорируется при создании IrSock-сервера, но важно для клиента, поскольку указывает на IrDA-устройство, к которому он подключен. (В зоне действия может быть несколько устройств.) И, наконец, поле *IrdaServiceName* — это имя службы, которую приложение регистрирует или к которой пытается подключиться.

### Разрешение имен

Адресация может быть основана на селекторах (IrDA Logical Service Access Point Selector, LSAP-SEL) или на службах, зарегистрированных Information

Access Services (IAS) IAS абстрагирует службу от LSAP-SEL в виде дружественного текстового имени, примерно так же, как DNS-сервер отображает имена компьютеров в цифровые IP-адреса. Для успешного соединения вы можете использовать как LSAP-SEL, так и дружественное пользователю имя, но в последнем случае требуется разрешать имена. Как правило, прямой LSAP-SEL-«адрес» не применяют, поскольку адресное пространство служб IrDA ограничено. Реализация Win32 разрешает использовать целые идентификаторы LSAP-SEL в диапазоне 1-127. По сути, IAS-сервер можно рассматривать как WINS-сервер, поскольку он ассоциирует LSAP-SEL с текстовым именем службы.

Фактически в записи IAS для нас важны лишь три поля: *имя класса* (class name), *атрибут* (attribute) и *значение атрибута* (attribute value). Допустим, сервер хочет регистрироваться под именем службы *MyServer*. Для этого ему нужно осуществить привязку к соответствующей структуре *SOCKADDR\_IRDA*. Затем добавляется запись IAS с именем класса *MyServer*, атрибутом *IrDA TinyTP LsapSel* и значением атрибута, например, 3. Это значение атрибута — следующий неиспользуемый LSAP-SEL, назначенный системой после регистрации. С другой стороны, клиент передает структуру *SOCKADDR\_IRDA* вызову подключения. В результате IAS начинает искать службу с именем класса *MyServer* и атрибутом *IrDA TinyTP LsapSel*. Этот IAS-запрос вернет значение 3. Вы можете сформулировать свой IAS-запрос с помощью параметра *IRLMP\_IAS\_QUERYB* вызове *getsockopt*.

Если вы хотите полностью проигнорировать IAS (что не рекомендуется), задайте LSAP-SEL-адрес напрямую для имени сервера или конечной точки, с которой хочет соединиться клиент. Обходить IAS, требуется только при подключении к устаревшим IrDA-устройствам, которые не поддерживают IAS-регистрацию (например, принтерам с инфракрасным портом). Для обхода IAS-регистрации и поиска задайте имя службы в структуре *SOCKADDR\_IRDA* как LSAP-SEL-xxx, где xxx — значение атрибута в диапазоне 1-127. В итоге серверу будет явно назначен данный LSAP-SEL-адрес, а клиент, не ведя IAS-поиск, сразу попытается подключиться к любой службе, выполняемой на LSAP-SEL.

## Нумерация IrDA-устройств

Поскольку инфракрасные (ИК) устройства появляются и исчезают из радиуса связи, необходим метод динамического перечисления соседних устройств. Реализация этого механизма в Windows CE и в Windows 98/2000 несколько отличается. В Windows CE поддержка IrSock появилась раньше, чем в Windows 98 и Windows 2000, но последние в ответ на запрос нумерации возвращают дополнительную «справочную» информацию. Поэтому заголовочный файл *Af\_irda.h* для Windows CE содержит исходные минимальные определения структур, а новые заголовочные файлы других платформ — Дополнительные определения для каждой платформы, поддерживающей rsock. Для согласованности рекомендуется использовать более поздние версии заголовочного файла *Af\_irda.h*.

Нумерация соседних ИК-устройств выполняется функцией *getsockopt* с параметром *IRLMP\_ENUM\_DEVICES* для Структура *DEVICEUST* передается как

параметр *optval*. Существуют две структуры, одна для Windows 98 и Windows 2000, а другая — для Windows CE. Они определены так:

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                numDevice;
    WINDOWS_IRDA_DEVICE_INFO Device[1];
} WINDOWS_DEVICELIST, «PWINDOVS_DEVICELIST, FAR *LPWINDOWS_DEVICELIST;
```

```
typedef struct _WCE_DEVICELIST
```

```
{
    ULONG                numDevice;
    WCE_IRDA_DEVICE_INFO Device[1];
} WCE_DEVICELIST, *PWCE_DEVICELIST;
```

Единственное отличие между ними — структура для Windows 98 и Windows 2000 содержит массив структур *WINDOWS\_IRDA\_DEVICE\_INFO* вместо *WCE\_IRDA\_DEVICE\_INFO*. Условная директива *\*define* объявляет *DEVICEUST* как соответствующую структуру, в зависимости от целевой платформы. Аналогично, существует и два объявления структуры *IRDA\_DEVICE\_INFO*.

```
typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char irdaDevlceID[4];
    char   irdaDevlceName[22];
    u_char irdaOeviceHints1;
    u_char irdaDeviceHints2;
    u_char irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOVS_IRDA_DEVICE_INFO,
FAR *LPWINDOWS_IRDA_DEVICE_INFO;
```

```
typedef struct _WCE_IRDA_OEVICE_INFO
{
    u_char irdaDeviceID[4];
    char   irdaDeviceName[22];
    u_char Reserved[2];
} WCE_IRDA_DEVICE_INFO, *PWCE_IRDA_DEVICE_INFO;
```

Условная директива *\*define* объявляет *IRDA\_DEVICE\_INFO*, опять-таки, в зависимости от платформы.

Как мы уже говорили, фактически нумерация ИК-устройств выполняется функцией *getsockopt* с параметром *IRLMP\_ENUM\_DEVICES*. Следующий код перечисляет идентификаторы всех ИК-устройств по соседству.

```
SOCKET    sock;
DEVICELIST devList;
DWORD      dwListLen=sizeof(DEVICELIST);

sock = WSASocket(AF_IRDA, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);

devList.numDevice = 0;
```

```
dwRet = getsockopt(sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
    (char *)&devList, AdwHstLen);
```

Перед тем как структура *DEVICELIST* будет передана в вызове *getsockopt*, не забудьте присвоить полю *numDevice* значение 0. При успешной нумерации полю *numDevice* будет присвоено положительное значение, равное числу структур *IRDADEVICEINFO* в поле *Device*. В реальном приложении выполнять *getsockopt* придется неоднократно, чтобы отследить все устройства, попавшие в радиус связи. Например, программа должна пытаться обнаружить ИК-устройство не менее пяти раз. Для этого просто поместите вызов процедуры в цикл с коротким вызовом функции *Sleep* после каждой безуспешной нумерации.

Зная, как нумеровать ИК-устройства, создать клиент или сервер не сложно. Серверная часть немного проще, поскольку выглядит как «обычный» сервер, то есть никаких особых действий не требует. Вкратце порядок действия IrSock-сервера таков.

1. Создание сокета с семейством адресов *AFJRDA* и типом *SOCK\_STREAM*.
2. Запись в структуру *SOCKADDRJRDA* имени службы сервера.
3. Вызов функции *bind* с описателем сокета и структурой *SOCKADDRJRDA*.
4. Вызов функции *listen* с описателем сокета и тайм-аутом.
5. Блокирование вызова функции *accept* для входящих запросов клиентов.

Действия клиента сложнее, поскольку необходимо нумеровать ИК-устройства.

1. Создание сокета с семейством адресов *AFJRDA* и типом *SOCKSTREAM*.
2. Нумерация доступных ИК-устройств путем вызова *getsockopt* с параметром *IRLMP\_ENUMDEVICES*.
3. Для каждого найденного устройства в структуру *SOCKADDRJRDA* — запись идентификатора найденного устройства, а также имени службы, к которой вы хотите подключиться.
4. Вызов функции *connect* с описателем сокета и структурой *SOCKADDRJRDA*. Это действие выполняется для каждой структуры, обработанной на шаге 3-

## Опрос IAS

Существует два способа определить, запущена ли данная служба на конкретном устройстве. Первый — попытаться подключиться к службе, другой — запросить у IAS имя службы. Оба способа требуют нумерации всех ИК-устройств и попыток запросить каждое устройство, пока одно из них не ответит (или подключиться к нему). Запрос выполняется функцией *getsockopt* с параметром *IRIMPIAS\_QUERY*. Указатель на структуру *IAS\_QUERY* передается в параметре *optval*. И снова, существуют две структуры *IAS\_QUERY*: одна — для Windows 98 и Windows 2000, а другая — для Windows CE. Вот их описания.

```
typedef struct WINDOWSIAS_QUERY
```

```
    u_char    irdaDeviceID[4];    /*
```

```
    */
```



```

char        irdaClassName[IAS_MAX_CLASSNAME];
char        irdaAttribName[IAS_MAX_ATTRIBNAME];
u_long      irdaAttribType;
union

    LONG      irdaAttribLnt;
    struct
    {
        u_long      Len;
        u_char      OctetSeq[IAS_MAX_OCTET_STRING];
    } irdaAttribOctetSeq;
    struct
    {
        u_long      Len;
        u_long      CharSet;
        u_char      UsrStr[IAS_MAX_USER_STRING];
    } irdaAttribUsrStr;
} irdaAttribute;
} WINDOWS_IAS_QUERY, *PWINDOVS_IAS_QUERY,
  FAR «LPWINDOVS.IAS.QUERY;

typedef struct _WCE_IAS_QUERY

    u_char      irdaDeviceID[4];
    char        irdaClassName[61];
    char        irdaAttribName[61];
    u_short     irdaAttribType;
    union

        int      irdaAttribLnt;
        struct

            int      Len;
            u_char      OctetSeq[1];
            u_char      Reserved[3];
        } irdaAttribOctetSeq;
        struct

            int      Len;
            u_char      CharSet;
            u_char      UsrStr[1];
            u_char      Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;

```

ИТО HKJJ

Как видите, описания одинаковы, кроме длины некоторых символьных массивов.

Узнать LSAP-SEL-номер определенной службы просто: присвойте полю *irdaClassName* строку свойств для *LSAP-SELS*: IrDAIrLMP:LsapSel, а полю *irda-*

*AttributeName* — имя запрашиваемой службы. Кроме того, укажите в поле *irdaDeviceID* действительный код устройства в радиусе связи.

## Создание сокета

Поскольку IrSock поддерживает только потоки с установлением соединения, чтобы создать ИК-сокет, необходимо указать лишь несколько параметров. Зот, например, как создается ИК-сокет с помощью функций *socket* или *WSA-Socket*. Для Windows CE (из-за ограничений Winsock 1.1) используйте функцию *socket*.

```
s = socket(AF_IRDA, SOCK_STREAM, 0);  
s = WSASocket(AF_IRDA, SOCK_STREAM, 0, NULL, 0,  
             WSA_FLAG_OVERLAPPED);
```

Для определенности вы вправе передать *IRDA\_PROTO SOCK\_STREAM* в качестве параметра протокола. Впрочем, он не обязателен, поскольку в каталоге транспортов есть только одна запись семейства адресов *AF\_IRDA*. Значение *AF\_IRDA* в вызове заставляет по умолчанию использовать эту запись каталога транспортов.

## Параметры сокета

Большинство *SO\_*-параметров сокета не применимы к IrDA. поддерживает лишь *SO\_LINGER*. Специфичные для IrSock параметры сокета поддерживаются только сокетами семейства адресов *AF\_IRDA*. (Они обсуждаются в главе 9, посвященной параметрам сокета и их характеристикам).

## Протоколы IPX/SPX

Протокол Internetwork Packet Exchange (IPX) используется компьютерами с клиент-серверными сетевыми службами NetWare фирмы Novell. IPX обеспечивает связь без установления соединения между двумя процессами, следовательно, при передаче пакета рабочей станцией не гарантируется, что он достигнет пункта назначения. Если приложению требуется гарантировать доставку данных, причем именно по протоколу IPX, можно задействовать протокол более высокого уровня — например, Sequence Packet Exchange (SPX) или SPX II, в которых SPX-пакеты передаются при помощи IPX. Winsock позволяет приложениям связываться по IPX под управлением Windows 95, Windows 98, Windows NT и Windows 2000, но не Windows CE.

## Адресация

В IPX-сетях сегменты соединяются через IPX-маршрутизаторы. Каждому сегменту назначается уникальный четырехбайтный *номер сети* (network number). Эти номера используются IPX-маршрутизаторами для управления подключениями между разными сегментами сети. Компьютер, подключенный к сегменту сети, идентифицируется при помощи шестибайтного *номера узла* (node number) — обычно это физический адрес сетевого адаптера. Узел

(компьютер) вправе запускать несколько процессов связи по IPX, для различения которых применяются номера сокетов

Для подготовки Winsock-клиента или сервера к подключению по IPX, необходимо настроить структуру *SOCKADDR\_IPX*. Она определена в заголовочном файле *Wsipx.h*, ссылка на него в приложении должна идти вслед за *Winsock2.h*. Структура *SOCKADDR\_IPX* определена так

```
typedef struct sockaddr_ipx
{
    short          sa_family,
    char           sa_netnum[4],
    char           sa_nodenum[6],
    unsigned short sa_socket,
} SOCKADDR_IPX, *PSOCKADDR_IPX, FAR *LPSOCKADDR_IPX,
```

Поле *sa\_family* всегда равно *AF\_IPX*. Поле *sa\_netnum* — 4-байтный номер сети в сегменте IPX-сети. Поле *sa\_nodenum* — 6-байтный номер узла. Поле *sa\_socket* представляет сокет или порт, используемый для различения IPX-подключений на одном узле.

## Создание сокета

Создать IPX-сокет можно несколькими способами. Для открытия IPX-сокета вызовите функции *socket* или *WSASocket* с семейством адресов *AF\_IPX*, типом сокета *SOCK\_DGRAM* и протоколом *NSPROTO\_IPX*.

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX),
s = WSASocket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX,
              NULL, 0, WSA_FLAG_OVERLAPPED),
```

Заметьте, что третий параметр протокола должен быть обязательно задан и не равен 0. Это важно, поскольку поле протокола может использоваться для настройки особых типов IPX-пакетов.

Как мы уже говорили, IPX обеспечивает ненадежное соединение без дейтаграмм. Если приложению требуется надежное соединение с применением этого протокола, оно может использовать поверх IPX протоколы более высокого уровня, такие как SPX и SPX II. Для этого необходимо при вызове функций *socket* или *WSASocket* задать соответствующие значения полей типа — *SOCK\_SEQPACKET* или *SOCK\_STREAM*, и протокола — *NSPROTO\_SPX* или *NSPROTO\_IPXII*.

Если тип сокета — *SOCK\_STREAM*, данные передаются в виде непрерывного потока байт, без разделения сообщений, подобно действию сокета в TCP/IP. Если тип сокета — *SOCK\_SEQPACKET*, данные передаются с разделителями сообщений. Например, передатчик отправляет 2000 байт — приемник не сможет ответить, пока не получит все 2000 байт. В SPX и SPXII для этого нужно указать бит конца сообщения в заголовке SPX. Для сокетов типа *SOCK\_SEQPACKET* подразумевается, что этот бит указан, и выполнение Winsock-функции *recv* и *WSARecv* не завершится, пока пакет не будет получен. Для поточных сокетов бит конца сообщения не требуется, и выполнение функции *recv* будет завершено сразу же по получении любых данных, независи-

мо от наличия бита конца сообщения С точки зрения отправителя при использовании типа *SOCK\_SEQPACKET* сообщения размером меньше пакета, всегда передаются с указанным битом конца При отправке нескольких пакетов этот бит задается только в последнем пакете

### Привязка сокета

Когда IPX-приложение при помощи функции *bind* создаст привязку локального адреса к сокету, указывать номер сети и адрес узла в структуре *SOCKADDR\_IPX* не нужно Функция *bind* заполнит эти поля при помощи первого же сетевого интерфейса IPX, доступного в системе Если на компьютере установлено несколько сетевых адаптеров, привязка к конкретному интерфейсу также не требуется В Windows 95, Windows 98, Windows NT и Windows 2000 реализована виртуальная внутренняя сеть, в которой к каждому сетевому адаптеру можно обратиться, независимо от того, к какой физической сети он подключен (Внутренние номера сети подробно рассмотрены далее в этой главе) После успешной привязки приложения к локальному интерфейсу с помощью функции *getsockname* вы можете узнать номер локальной сети и номер узла следующим образом

```
SOCKET sdServer,
SOCKADDR_IPX IPXAddr,
int addrlen = sizeof(SOCKADDR_IPX),

if ((sdServer = socket (AF_IPX, SOCK_DGRAM, NSPROTO_IPX))
    == INVALID_SOCKET)
{
    printf( socket failed with error %d\n ,
           WSAGetLastError0),
    return,

ZeroMemory(&IPXAddr, sizeof(SOCKADDR_IPX)),
IPXAddr sa_family = AF_IPX,
IPXAddr sa_socket = htons(5150),

if (bind(sdServer, (PSOCKADDR) &IPXAddr, sizeof(SOCKADDR_IPX))
    == SOCKET_ERROR)
{
    printf( bind failed with error %d\n ,
           WSAGetLastError0),
    return,

(getsockname((unsigned) sdServer, (PSOCKADDR) &IPXAddr, &addrlen)
    == SOCKET_ERROR)

printf( getsockname failed with error %d\n , * '
       WSAGetLastError0),
return,
```

```
// Вывод информации SOCKADDR_IPX, возвращенной getsockname()
```

### Внутренний номер сети

В IPX номер сети (внутренний или внешний) определяет сегменты сети и применяется для маршрутизации IPX-пакетов между сегментами. В Windows 95, Windows 98, Windows NT и Windows 2000 предусмотрен внутренний номер сети, используемый для внутренней маршрутизации и четкой идентификации компьютера при межсетевых подключениях (несколько сетей, соединенных мостами). Внутренний номер сети также называется виртуальным, поскольку определяет еще один (виртуальный) сегмент межсетевого соединения. Таким образом, при настройке внутреннего номера сети для компьютеров под управлением Windows 95, Windows 98, Windows NT или Windows 2000 сервер NetWare или IPX-маршрутизатор добавляют дополнительный транзит (hop) в маршрут к этому компьютеру.

На компьютере с несколькими сетевыми адаптерами внутренняя виртуальная сеть выполняет специальную задачу. Для привязки к локальному сетевому интерфейсу приложению не надо указывать информацию о локальном интерфейсе: достаточно присвоить значение 0 полям *sajnetnum* и *sanodenum* структуры *SOCKADDR\_IPX*. Дело в том, что IPX может маршрутизировать пакеты из любой внешней сети к любому локальному сетевому интерфейсу при помощи внутреннего номера сети. Даже если приложение явно привязано к сетевому интерфейсу в сети А, а пакет пришел по сети В, внутренний номер сети все-таки позволит передать его приложению.

### Установка типа IPX-пакета средствами Winsock

Выбрать тип IPX-пакета можно при создании сокета, задав параметр *NSPROTO\_IPX*. Поле типа пакета в IPX-пакете указывает тип службы, предложенной или запрошенной IPX-пакетом. Фирмой Novell определены следующие типы пакетов:

- III 01h — Routing Information Protocol (RIP),
- III 04h — Service Advertising Protocol (SAP),
- III 05h — Sequenced Packet Exchange (SPX),
- M 11h — NetWare Core Protocol (NCP),
- 14h — широковещательный пакет для Novell NetBIOS

Чтобы изменить типа IPX-пакета, достаточно просто указать *NSPROTO\_IPX + n* в качестве параметра протокола в функции *socket*, где *n* — номер типа пакета. Например, для открытия IPX-сокета с типом пакета 04h (SAP), вызовите *socket* так:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX + 0x04);
```

## разрешение имен

ipX-адресация в Winsock довольно неудобна, поскольку для формирования адреса нужно предоставить многобайтовые номера сети и узла. Впрочем, IPX позволяет приложениям обнаруживать службы путем использования дружественных имен для получения номера сети, узла и порта по протоколу SAP. Winsock 2 предоставляет независимый от протокола способ регистрации имен функцией *WSASetService* (подробней — в главе 10). По протоколу SAP IPX-сервер может задействовать эту функцию для регистрации под дружественным именем номеров сети, узла и порта, которые слушает Winsock. 2 также предоставляет независимый от протокола способ разрешения имен с помощью функций *WSALookupServiceBegin*, *WSALookupServiceNext* и *WSALookupServiceEnd*.

Можно выполнить свою собственную регистрацию <имя — служба> и вести поиск, открыв IPX-сокеты и выбрав тип SAP пакета. После открытия сокета вы вправе начать широковещание SAP-пакетов в IPX-сети для регистрации и обнаружения в ней служб. Но для этого необходимо хорошо знать протокол SAP и уметь программно декодировать SAP-пакет IPX.

## Протоколы NetBIOS

Этот интерфейс уже рассматривался в главе 1, поэтому обсуждаемый здесь материал покажется вам знакомым. Для адресации NetBIOS из Winsock необходимо знать имена NetBIOS и номера LAN. Здесь мы уделим основное внимание особенностям доступа к NetBIOS из Winsock.

**ПРИМЕЧАНИЕ** Семейство адресов NetBIOS доступно в Winsock только под управлением Windows NT и Windows 2000. На платформах Windows 9x и Windows CE оно недоступно.

## Адресация

Адресация компьютеров в NetBIOS основана на именах NetBIOS. Напомним, что имя NetBIOS состоит из 16 символов, причем последний обозначает тип службы, к которой относится это имя. Имена NetBIOS бывают уникальными и групповыми. Уникальное имя может использоваться только одним процессом в сети. Например, если сеансовый сервер регистрируется под именем FOO, то для соединения с ним клиенты будут использовать это имя. Под групповым именем может регистрироваться группа приложений — тогда направляемые на это имя дейтаграммы получают все зарегистрировавшие его процессы.

Структура адресации NetBIOS в Winsock определена в файле `Wsnetbs.h`

```
«define NETBIOS_NAME_LENGTH 16
```

```
    struct sockaddr_nb
```

```
    short    snb_family,
    "short    snb_type,
```

```

    char    snb_name[NETBIOS_NAME_LENGTH];
} SOCKADDR_NB, *PSOCKADDR_NB, FAR *LPSOCKADDR_NB;

```

Поле *snbfamily* задает семейство адресов этой структуры, поэтому оно всегда должно быть равно *AF\_INET*. В поле *snbjtype* указывается тип имени: уникальное или групповое. Для этого поля можно использовать следующие определения:

```

<define NETBIOS_UNIQUE_NAME      (0x0000)
<define NETBIOS_GROUP_NAME      (0x0001)

```

Наконец, в поле *snbjname* содержится собственно имя NetBIOS.

Зная, что означает каждое поле и чему оно должно быть равно, вы можете разобраться в следующем полезном макросе, который определен в заголовочном файле и задает нужные значения полям данной структуры:

```

<define SET_NETBIOS_SOCKADDR(_snb, „type, _name, _port)

    int _i;
    (_snb)->snb_family = AF_INET;
    (_snb)->snb_type = (_type);
    for (_i = 0; _i < NETBIOS_NAME_LENGTH - 1;
        (_snb)->snb_name[_i] = ' ');
    }
    for (_i = 0;
        •((.name) + _i) != '\\0'
        && _i < NETBIOS_NAME_LENGTH - 1;

        (_snb)->snb_name[_i] = *((_name)+_i);
    }
    (_snb)->snb_name[NETBIOS_NAME_LENGTH - 1] = (_port);

```

Первый параметр макроса — *\_snb*, адрес заполняемой вами структуры *SOCKADDR\_NB*. Как видите, полю *snbfamily* автоматически присваивается значение *AF\_INET*. Для параметра *Jtype* задайте значение *NETBIOS\_UNIQUE\_NAME* или *NETBIOS\_GROUP\_NAME*. Параметр *jname* — это имя NetBIOS; предполагается, что оно состоит либо из *NETBIOS\_NAME\_LENGTH - 1* символов, либо содержит меньшее число символов и является строкой с 0 в конце. Заметьте, что поле *snbname* сначала заполняется пробелами, а в конце макроса 16-й символ строки *snbjname* получает значение *jport*.

Как видите, определить структуру имени NetBIOS в Winsock достаточно просто. В отличие от TCP и IrDA, разрешение имени в NetBIOS скрыто от вас, поэтому не нужно сопоставлять имени физический адрес перед началом работы. Дело в том, что NetBIOS использует несколько протоколов в качестве протоколов нижнего уровня, и каждый из них имеет собственную схему адресации. В следующем разделе мы приведем пример простого клиент-серверного приложения, использующего интерфейс NetBIOS в Winsock.

## Создание сокета

При создании NetBIOS-сокета очень важно верно задать номер LANA. Как и для собственного API NetBIOS, нужно знать, какие номера LANA доступны приложению. Помните, что клиент и сервер NetBIOS должны использовать общий транспортный протокол для прослушивания и соединения. Существует два способа создания сокета NetBIOS. Первый — вызвать функцию *socket* или *WSASocket*:

```
s = WSASocket(AF_NETBIOS, SOCK_DGRAM | SOCK_SEQPACKET, -lana,
              NULL, 0, WSA_FLAG_OVERLAPPED);
```

Чтобы использовать дейтаграммный сокет, назначьте параметру *type* функции *WSASocket* значение *SOCK\_DGRAM*, а сокету с установлением соединения — значение *SOCK\_SEQPACKET* (но не оба одновременно). Третий параметр *protocol*, номер LANA, на котором нужно создать сокет (он должен быть отрицательным). Значение четвертого параметра — *NULL*, так как вы задаете свои собственные параметры, не используя структуру *WSAPROTOCOL\_INFO*. Пятый параметр не используется. Наконец, параметр *dwFlags* равен *WSA\_FLAG\_OVERLAPPED*. Это значение следует использовать при всех обращениях к функции *WSASocket*.

Недостаток этого способа создания сокетов в том, что вы должны знать доступные номера LANA. К сожалению, в Winsock не существует простого способа нумерации LANA. Конечно, для выяснения свободных номеров LANA можно использовать функцию *Nethios* с параметром *NCBENUM*. Но Winsock предлагает альтернативу — перечисление всех транспортных протоколов с помощью функции *WSAEnumProtocols* (см. главу 5). В следующем примере перечисляются все транспортные протоколы, обнаруживаются транспорты NetBIOS и создаются сокеты для каждого из них.

```
dwNum = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);
if (dwNum == SOCKET_ERROR)
{
    // Ошибка
}
for (i = 0; i < dwNum; i++)
{
    // Поиск записей в семействе адресов AF_NETBIOS
    if (lpProtocolBuf[i].iAddressFamily == AF_NETBIOS)
    {
        // поиск сокетов с типом SOCK_SEQPACKET или SOCK_DGRAM
        if (lpProtocolBuf[i].iSocketType == SOCK_SEQPACKET)
        {
            = WSASocket(FROM_PROTOCOL_INFO,
                        FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
                        &lpProtocolBuf[i], 0, WSA_FLAG_OVERLAPPED);
```



В данном псевдокоде все доступные протоколы сначала нумеруются, а затем в цикле выявляются те, которые относятся к семейству адресов *AF^NET-BIOS*. Затем идет поиск сокетов с типом *SOCK\_SEQPACKET*. Если бы мы хотели передавать дейтаграммы, следовало бы искать сокет с типом *SOCK\_DGRAM*. Когда тип сокета совпадет с желаемым — найден доступный для использования транспорт NetBIOS. Если нужен номер LANA, возьмите абсолютное значение поля *iProtocol* структуры *WSAPROTOCOLINFO*. Единственное исключение — номер 0. Поле *iProtocol* для этого LANA равно 0x80000000, так как номер 0 зарезервирован для Winsock. Количество действительных портов содержится в переменной/

## Протокол AppleTalk

Winsock поддерживает AppleTalk уже давно, хотя знают об этом немногие. Вероятнее всего, вы не захотите использовать протокол AppleTalk, если только вам не нужно соединиться с компьютерами Macintosh. AppleTalk похож на NetBIOS: сервер динамически регистрирует определенное имя, по которому с ним могут соединяться клиенты. Впрочем, имена AppleTalk значительно сложнее имен NetBIOS

## Адресация

Имя AppleTalk фактически основано на трех отдельных именах: собственно имени, типе и зоне. Длина каждого из этих имен не может быть больше 32 символов. Имя идентифицирует процесс и связанный с ним сокет на компьютере. Тип — это механизм группировки для зон. Обычно зона представляет собой сеть компьютеров, поддерживающих AppleTalk и расположенных в одной *петле* (loop). Реализация протокола AppleTalk от Microsoft позволяет компьютеру под управлением Windows указать зону, в которой он находится. Несколько сетей можно соединить мостами. Номеру сокета, узла и сети соответствуют дружественные имена. Протокол Name Binding Protocol (NBP) требует, чтобы имя AppleTalk было уникальным для данного типа и зоны. Для проверки уникальности имени этот протокол применяет широкоэвещательные запросы. Чтобы динамически определить маршруты к соединенным сетям, AppleTalk использует Routing Table Maintenance Protocol (RTMP).

В основе адресации узлов AppleTalk из Winsock лежит структура:

```
typedef struct sockaddr_at
{
    USHORT  sat_family;
    USHORT  sat_net;
    UCHAR   sat_node;
    UCHAR   sat_socket;
} SOCKADDR_AT, *PSOCKADDR_AT;
```

Заметьте: эта структура содержит только символы и короткие целые числа, но не дружественные имена. Структура *SOCKADDR\_AT* передается в таких функциях Winsock, как *bind*, *connect* и *WSAConnect*, однако для трансляции

пужественного имени необходимо сначала разрешить или зарегистрировать это имя функциями *getsockopt* и *setsockopt* соответственно.

## Регистрация имени AppleTalk

Зарегистрировать сервер под определенным именем, которое будут использовать клиенты, позволяет функция *setsockopt* с параметром *SO\_REGISTER\_NAME*. Для всех параметров сокетов, связанных с именами AppleTalk, используйте структуру *WSH\_NBP\_NAME*:

```
typedef struct
{
    CHAR      ObjectNameLen;
    CHAR      ObjectName[MAX_ENTITY];
    CHAR      TypeNameLen;
    CHAR      TypeName[MAX_ENTITY];
    CHAR      ZoneNameLen;
    CHAR      ZoneName[MAX_ENTITY];
} WSH_NBP_NAME, «PWSH_NBP_NAME;
```

Ряд типов: например, *WSH\_REGISTER\_NAME*, *WSH\_DEREGISTER\_NAME* и *WSH\_REMOVE\_NAME*, — определены на основе структуры *WSH\_NBP\_NAME*. Выбор типа зависит от того, ищите ли вы имя, регистрируетесь под ним, или удаляете его.

Вот как зарегистрировать имя AppleTalk:

```
«define MY_ZONE
«define MY_TYPE      "Winsock-Test-App"
«define MY_OBJECT     "AppleTalk-Server"

WSH_REGISTER_NAME    atname;
SOCKADDRfi_AT        ataddr;
SOCKET               s;

// Впишите регистрируемое имя
//
strcpy(atname.ObjectName, MY_OBJECT);
atname.ObjectNameLen = strlen(MY_OBJECT);
strcpy(atname.TypeName, MY_TYPE);
atname.TypeNameLen = strlen(MY_TYPE);
strcpy(atname.ZoneName, MY_ZONE);
atname.ZoneNameLen = strlen(MY_ZONE);

s = socket(AF_APPLETALK, SOCK_STREAM, IPPROTO_UDP);
if (s == INVALID_SOCKET)

    // Ошибка

ataddr.sat^socket = O-
a ? at_family = AF-
(s, (SOCKADDR *)&ataddr, sizeof(at*Alie5/r« SOCKET_ERROR)
```

```
// Невозможно открыть конечную точку в сети AppleTalk
}
if (setsockopt(s, SOL_APPLETALK, SO_REGISTER_NAME,
              (char O&atname, sizeof(WSH_NBP_NAME)) == SOCKET_ERROR)
{
    // Ошибка при регистрации имени1
```

Первое, на что следует обратить внимание, — строки *MYZONE*, *MYJTYPE* *YL MY OBJECT*. Как вы помните, имя AppleTalk трехуровневое. В рассматриваемом примере вместо имени зоны стоит звездочка (\*). Она обозначает текущую зону, то есть зону, в которой находится ваш компьютер. Затем создается сокет с типом *SOCK\_STREAM* и вызывается функция *bind* со структурой адреса, где значение поля *sat* сокет обнулено и только полю семейства протоколов присвоено значение. Это важно, поскольку в результате в сети AppleTalk создается конечная точка, откуда ваше приложение сможет выполнять запросы. Хотя вызов функции *bind* и позволяет выполнять простейшие операции в сети, сам по себе он не дает возможности приложению принимать входящие клиентские запросы. Для этого необходимо сделать следующий шаг — зарегистрировать имя в сети.

Сделать это не так уж сложно: вызовите функцию *setsockopt*, передав в нее в качестве параметра *level* *SOL\_APPLETALK*, а в качестве параметра *optname* — *SO\_REGISTER\_NAME*. Эти параметры — указатели на структуру *WSHJREGISTERJSIAME* и ее размер. Успешный вызов данной функции означает, что имя сервера было зарегистрировано. Если при вызове произошла ошибка, вероятнее всего, имя уже используется кем-то другим. Код этой ошибки Winsock — *WSAEADDRINUSE* (10048 или 0x02740h). Заметьте, что для получения данных процесс должен зарегистрировать имя, вне зависимости от того, обмениваетесь ли вы дейтаграммами или устанавливаете соединение.

## Разрешение имен AppleTalk

Клиентское приложение обычно знает дружественное имя сервера и для вызова функций Winsock должно разрешить его в номер сети, узла и сокета. Эта задача решается путем вызова функции *getsockopt* с параметром *SO\_LOOKUP\_NAME*. Для поиска имени применяется структура *WSH\_LOOKUP^NAME*, а также используемая в ней структура *WSH\_NBP\_TUPLE*.

```
typedef struct
```

```
    WSH_ATALK_ADDRESS    Address;    f_ЛЗЯЧТЛ ,ir    . , , <
    USHORT              Enumerator;
    WSH_NBP_NAME         NbpName;
} WSH_NBP_TUPLE, *PWSH_NBP_TUPLE;
```

```
typedef struct _WSH_LOOKUP_NAME
```

```
// Массив структур WSH_NBP_TUPLE    ж*лш .1Б»ъч,г>{(- ;W)XiS/
WSH_NBP_TUPLE    LookupTuple;
```

```

        ULONG                NoTuples;
> ws_H_LOOKUP_NAME, *WSH_LOOKUP_NAME;                                f

```

При вызове функции *getsockopt* с параметром *SO\_LOOKUP\_NAME* мы передаем в нее в качестве буфера структуру *WSH\_LOOKUP\_NAME*, а в поле *\\S#\_BIBP\_NAME* — задаем первый член массива *LookupTuple*. В случае успешного вызова функция *getsockopt* возвращает массив элементов *WSHJVBP\_TUPLE*, содержащих информацию о физическом адресе для данного имени. Поиск имени проиллюстрирован в листинге 6-1 (файл *Atalknmc*). Кроме того, в примере показано, как перечислить все «обнаруженные» зоны AppleTalk и найти текущую зону. Информация о зонах содержится в параметрах *SO\_LOOKUP\_ZONES* и *SO\_LOOKUP\_MYZONE* функции *getsockopt*.

### Листинг 6-1. Поиск имени и зоны AppleTalk

```

<include <winsock.h>
<include <atalkwsh.h>                                U][t

<include <stdio.h>
<include <stdlib.h>                                «•»

<define DEFAULT_ZONE
<define DEFAULT_TYPE        "Windows Sockets"
<define DEFAULT_OBJECT      "AppleTalk-Server"

char szZone[MAX_ENTITY],
      szType[MAX_ENTITY],
      szObject[MAX_ENTITY];

BOOL bFmdName = FALSE,
      bListZones = FALSE,
      bListMyZone = FALSE;

void usage()
<
    printf("usage. atlookup [options]\n");
    printf("        Name Lookup \n"),
    Printf("        -z-ZONE-NAME\n");                ;>
    printf("        -t TYPE-NAME\n");
    pnntfC    к    -o-OBJECT-NAME\n");
    pnntfC    List All Zones.\n");
    Pnntf("        -lz\n");
    printf("        List My Zone:\n");                ,>,<и и
    PnntfC    -lm\n");
    ExitProcessd);
}
(V

void ValidateArgsdnt argc, char "argv)

xnt
1;

```

Листинг 6-1. *(продолжение)*

```

strcpy(szZone, DEFAULT_ZONE);
strcpy(szType, DEFAULT_TYPE);
strcpy(szObject, DEFAULT_OBJECT);

for(i = 1; i < argc; i++)
{
    if (strlen(argv[i]) < 2)
        continue;
    if ((argv[i][0] == '-') || (argv[i][0] == 'V'))
    {
        switch (tolower(argv[i][1]))
        {
            case 'z':          // Указание имени зоны
                if (strlen(argv[i]) > 3)
                    strcpy(szZone, &argv[i][3], MAX_ENTITY);
                bFindName = TRUE;
                break;
            case 't':          // Указание имени типа
                if (strlen(argv[i]) > 3)
                    strcpy(szType, &argv[i][3], MAX_ENTITY);
                bFindName = TRUE;
                break;
            case 'o':          // Указание имени объекта
                if (strlen(argv[i]) > 3)
                    strcpyCszObject, &argv[i][3], MAX_ENTITY);
                bFindName = TRUE;
                break;
            case 'l':          // Просмотр информации о зонах
                if (strlen(argv[i]) == 3)
                    // List all zones
                    if (tolower(argv[i][2]) == 'z')
                        bListZones = TRUE;
                    // List my zone
                else if (tolower(argv[i][2]) == 'm')
                    bListMyZone = TRUE;
                break;
            default:
                usage();
        }
    }
}

int main(int a^b, char **argv)
{
    WSADATA wsd;
    'har J- cLookupBuffer[16000J,
    PWSH_NBP_TBR6 •pTupleBuffer = NULL;
    pTuples = NULL;

```

Листинг 6-1. (продолжение)

```

FWH_LOOKUP_NAME    atlookup;
FWH_LOOKUP_ZONES   zonelookup;
SOCKET              s;
DWORD              dwSize = sizeof(cLookupBuffer);
SOCKADDR            ataddr;
int                 i;

// Загрузка библиотеки Winsock

//
if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
<
    printf("Unable to load Winsock library!\n");
    return 1;

ValidateArgs(argc, argv);

atlookup = (FWH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (FWH_LOOKUP_ZONES)cLookupBuffer;
if (bFindName)

    // Ввод искомого имени
    //
    strcpy(atlookup->LookupTuple.NbpName.ObjectName, szObject);
    atlookup->LookupTuple.NbpName.ObjectNameLen =
        strlen(szObject);
    strcpy(atlookup->LookupTuple.NbpName.TypeName, szType);
    atlookup->LookupTuple.NbpName.TypeNameLen = strlen(szType);
    strcpy(atlookup->LookupTuple.NbpName.ZoneName, szZone);
    atlookup->LookupTuple.NbpName.ZoneNameLen = strlen(szZone);
}
// Создание сокета AppleTalk
//
s = socket(AF_APPLETALK, SOCK_STREAM, IPPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    printf("socket() failed: %d\n", WSAGetLastError());
    return 1;
}

// Для создания в сети AppleTalk конечной точки и выполнения запросов,
// необходимо осуществить привязку
ZeroMemory(&ataddr, sizeof(ataddr));
ataddr.sat_family = AF_APPLETALK;
ataddr.sat_socket = 0;
if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) ==
    INVALID_SOCKET)

```

*.imp**см. след. стр.*

Листинг 6-1. *(продолжение)*

```

printf("bind() failed: Xd\n", WSAGetLastErrorO);
return 1;

if (bFmdName)
{
    printf("Looking up %s:Xs@Xs\n", szObject, szType, szZone);
    if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_NAME,
        (char *)atlookup, &dwSize) == INVALID_SOCKET)
    {
        printf("getsockopt(SO_LOOKUP_NAME) failed: Xd\n",
            WSAGetLastErrorO);
        return 1;
    }
    >
    printf("Lookup returned: %d entries\n",
        atlookup->NoTuples);
    //
    // Символьный буфер содержит массив структур
    // WSH_NBP_TUPLE, вслед за структурой WSH_LOOKUP_NAME
    //
    pTupleBuffer = (char *)cLookupBuffer +
        sizeof(WSH_LOOKUP_NAME);
    pTuples = (WSH_NBP_TUPLE) pTupleBuffer;    <^

    for(i = 0;    atlookup->NoTuples;

        ataddr.sat.family = AF_APPLETALK,
        ataddr.sat.net     = pTuples[i].Address.Network;
        ataddr.sat_node    = pTuples[i].Address.Node,
        ataddr.sat_socket  = pTuples[i].Address.Socket;
        printf("server address = Xlx.Xlx Xlx.\n",
            ataddr.sat_net,
            ataddr.sat_node,
            ataddr.sat_socket);

    }
    else if (bListZones)                                JAV,-

        // Для этого параметра необходимо выделить достаточный размер буфера,
        // иначе в Windows NT 4 SP3 появляется ошибка "синяя смерть" (blue screen)
        //
        if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES,    om-
            (char *)atlookup, &dwSize) == INVALID_SOCKET)*
        (
            <<7
            printf("getsockopt(SO_LOOKUP_NAME) failed- Xd\n",    «e
                WSAGetLastErrorO);                                >
            return 1;                                            .JAVKA
        >
        printf("Lookup returned: %6 zones\n", zonelookup->NoZonee);

```

Листинг 6-1. {продолжение}

```

//
// II Бuffer содержит список разделенных нулями строк вслед
// за структурой WSH_LOOKUP_NAME
//
pTupleBuffer = (char OcLookupBuffer +
                sizeof(WSH_LOOKUP_ZONES);
for(i = 0; i < zoneLookup->NoZones; i++)
{
    printf("X3d: 'Xs'\n", i+1, pTupleBuffer);
    while (*pTupleBuffer++);

else if (bListMyZone)

// Этот параметр возвращает простую строку

if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_MZONE,
              (char OcLookupBuffer, &dwSize) = INVALID_SOCKET)
{
    printf("getsockopt(SO_LOOKUP_NAME) failed: Xd\n",
          WSAGetLastErrorO);
    return 1;
}
printf("My Zone: 'J(s'\n", cLookupBuffer);
}
else
    usage();

WSACleanup();

return 0;
}

```

Для большинства параметров сокетов AppleTalk (например, *SO\_LOOKUP\_MYZONE*, *SO\_LOOKUP\_ZONES* или *SO\_LOOKUP\_NAME*) при вызове функции *getsockopt* необходимо выделить достаточно большой символьный буфер. Если в качестве параметра требуется указать структуру, то она должна находиться в начале буфера. При успешном вызове функции *getsockopt* возвращаемые данные помещаются в буфер после этой структуры.

Рассмотрим раздел с *SO\_LOOKUP\_NAME* в листинге 6-1. Для вызова функции *getsockopt* используется переменная *cLookupBuffer*, являющаяся простым массивом символов. Сначала мы помещаем туда структуру *PWSH\_LOOKUP* \* с информацией об искомом имени. Затем передаем буфер в функцию *getsockopt*, после чего увеличиваем на 1 указатель на символьный буфер *runpTupleBuffer*<sup>8</sup> чтобы он указывал на символ, находящийся после структуры *WSH\_LOOKUP\_NAME*. Далее присваиваем этому указателю адрес структуры *PWSH\_*



*NBPJTUPLE*, так как возвращаемые в результате поиска имени данные — массив структур *WSH\_NBP\_TUPLE*. Зная начальное положение и тип записей, мы можем просмотреть весь массив. Подробнее о параметрах сокетов — в главе 9.

## Создание сокета

Для создания сокета можно использовать любую подходящую функцию Winsock, начиная с версии 1.1. Существует два способа задать базовые протоколы AppleTalk: использовать определение соответствующего протокола из файла *Atalkwshh*, либо вызвать функцию *WSAEnumProtocols* и поместить результаты в структуру *WSAPROTOCOLINFO*. В табл. 6-1 для каждого протокола AppleTalk приведены параметры, необходимые для создания сокета функциями *socket* или *WSASocket*.

**Табл. 6-1. Протоколы и параметры AppleTalk, семейство адресов AFAPPLETALK**

<u>Протокол</u>	<u>Тип сокета</u>	<u>Тип протокола</u>
MSAFD AppleTalk [ADSP]	<i>SOCK_RDM</i>	<i>ATPROTOADSP</i>
MSAFD AppleTalk [ADSP] [псевдопоток]	<i>SOCKSTREAM</i>	<i>ATPROTO_ADSP</i>
MSAFD AppleTalk [PAP]	<i>SOCKRDM</i>	<i>ATPROTO_PAP</i>
MSAFD AppleTalk [RTMP]	<i>SOCKJXRAM</i>	<i>DDPPROTO_RTMP</i>
MSAFD AppleTalk [ZIP]	<i>SOCKDGRAM</i>	<i>DDPPROTO_LP</i>

## Протокол АТМ

Winsock 2 в Windows 98 и Windows 2000 поддерживает протокол Asynchronous Transfer Mode (ATM), применяемый как в локальных, так и в глобальных сетях для высокоскоростной передачи данных любого типа (в том числе звуковой и видеoinформации). Протокол АТМ гарантирует *качество обслуживания* (quality of service, QoS) за счет *виртуальных соединений* (Virtual Connections, VC) в сети. Для установления виртуальных соединений в сети АТМ Winsock использует семейство адресов АТМ. Типичная сеть АТМ состоит из конечных точек (компьютеров), соединенных коммутаторами (рис. 6-1).

При программировании протокола АТМ необходимо учитывать несколько особенностей. Во-первых, АТМ — это фактически не протокол, а тип носителя. Иными словами, технология АТМ напоминает запись кадров Ethernet прямо в сеть Ethernet. Как и Ethernet, сеть АТМ не способна управлять *потоками* (flow control). Протокол АТМ предварительно устанавливает соединение и работает в режимах сообщений или потоков. Это означает, что если отправка данных замедлится, исходное приложение может переполнить локальный буфер. Аналогично, если приложение-получатель не будет считывать данные достаточно часто, произойдет переполнение буфера на стороне получателя и часть принятой информации будет утеряна. Для управления потоками вы вправе использовать протокол IP поверх АТМ. В этом случае приложения будут работать с семейством IP-адресов. Хотя протокол АТМ обладает определенными преимуществами по сравнению с IP — например,

допускает корневую многоадресную схему (см. главу 12) — при выборе протокола нужно учитывать специфику приложения.

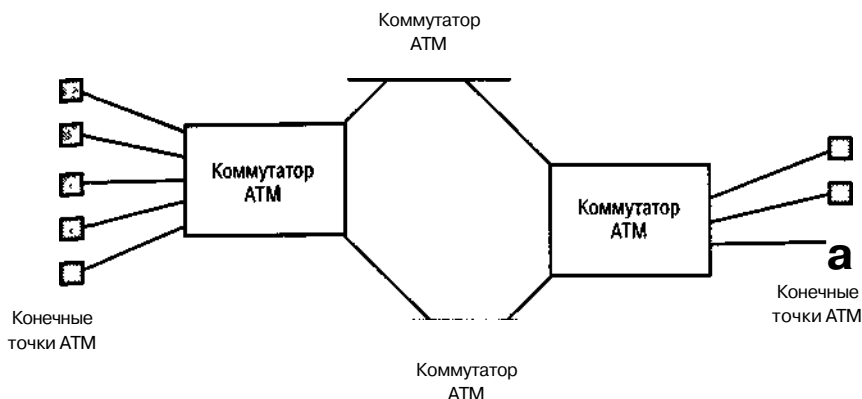


Рис. 6-1. Сеть ATM

**ПРИМЕЧАНИЕ** Так как поддержка ATM — новинка Winsock 2, информация из этого раздела протестирована только на реализации ATM для Windows 2000 Beta 3. Windows 98 (SP 1) не была доступна для тестирования ко времени написания этой книги. Поэтому содержащееся здесь описание протокола ATM может несколько отличаться от его окончательной реализации в Windows 2000 и Windows 98 (SP 1).

## Адресация

В сети ATM два сетевых интерфейса: «пользователь — сеть» (user network interface, UNI) и «узел — сеть» (network node interface, NNI). Первый определяет взаимодействие компьютеров и коммутаторов ATM, второй — взаимодействие двух коммутаторов. У каждого из этих интерфейсов есть соответствующий протокол.

- **Сигнальный протокол UNI** позволяет компьютеру и коммутатору ATM обмениваться информацией об установлении соединения и управляющей информацией, обеспечивая установку соединения в сети ATM. При использовании этого протокола компьютер может общаться только с ближайшим коммутатором, через несколько коммутаторов данные не передаются.

**Сигнальный протокол NNI** позволяет двум коммутаторам обмениваться информацией о маршрутизации и управляющей информацией.

Здесь описаны только те элементы сигнального протокола UNI, которые касаются установки соединения ATM средствами Winsock. В настоящее время Winsock в Windows 2000 и Windows 98 (SP 1) поддерживает сигнальный протокол UNI версии 3.1.

Для обмена данными по сети ATM с помощью сигнального протокола UNI, Winsock использует механизм *точек доступа к службам* (Service Access Point, SAP). Напомним, что для обмена данными через сеть ATM между компьютерами должно быть предварительно установлено виртуальное соединение. SAP позволяет приложениям Winsock зарегистрировать и идентифицировать интерфейс сокета для связи по сети ATM с помощью адресной структуры *SOCKADDR\_ATM*. Создав SAP, Winsock устанавливает виртуальное соединение между клиентом и сервером (используя сигнальный протокол UNI). Структура *SOCKADDR\_ATM* определена так:

```
typedef struct sockaddr_atm
```

```

    u_short      satm_family;
    ATMADDRESS   satm_number;
    ATM.BLLI     satm.blli;
    ATM.BHLI     satm.bhli;
    caddr_atm, SOCKADDR_ATM, *PSOCKADDR_ATM, *LPSOCKADDR_ATM;
```

Поле *satmfamily* всегда должно быть равно *AF\_ATM*. Поле *satmnumber* — фактический адрес ATM в виде структуры *ATM\_ADDRESS*. Он основан на одной из двух основных схем адресации ATM: E.164, либо *точках доступа к сетевым службам* (Network Service Access Point, NSAP). Адреса NSAP также называют *системными адресами ATM в стиле NSAP* (NSAP-style ATM Endsystem Address, AESA). Структура *SOCKADDR\_ATM* определена так:

```

typedef struct
{
    DWORD AddressType;
    DWORD NumofDigits;
    UCHAR Addr[ATM_ADDR_SIZE];
} ATM_ADDRESS;
```

В поле *AddressType* задается схема адресации: для E.164 значение поля должно быть *ATM\_E164*, для NSAP — *ATM\_NSAP*. Кроме того, если приложение пытается осуществить привязку сокета к SAP, то этому полю можно присвоить и другие значения:

- К *ATME164* — адрес E.164, применяется при соединении с SAP;
- III *ATMNSAP* — адрес в стиле NSAP, применяется при соединении с SAP;
- III *SAPFIELDANYAESASEL* — адрес в стиле NSAP с параметризованным селекторным октетом, служит для привязки сокета к SAP;
- *SAPFIELDANYAESAREST* — адрес в стиле NSAP со всеми октетами, кроме параметризованного, служит для привязки сокета к SAP.

Поле *NumofDigits* всегда должно быть равно *ATM\_ADDR\_SIZE*. В поле *Addr* содержится фактический 20-байтный адрес ATM по схеме E.164 или NSAP.

Поля *satm\_blli* и *satmbhli* структуры *SOCKADDR\_ATM* в ATM UNI представляют широкополосную информацию *нижнего* (Broadband Lower Layer Information, BLLI) и *верхнего уровня* (Broadband Higher Layer Information,

BHLI) соответственно. Вообще, эти структуры используются для идентификации стека протокола, работающего поверх соединения ATM. В документах по ATM Form/IETF описано несколько стандартных комбинаций значений BHLI и BLLI. (Одна комбинация идентифицирует соединение по ATM с эмуляцией ЛВС, другая — IP поверх ATM и т. п.) Полные диапазоны значений для полей этих структур приведены в книге стандартов ATM UNI 3.1, а документы по ATM Form/IETF можно найти по адресу <http://www.ietf.org>.

Структуры данных BHLI и BLLI определены так:

```
typedef struct
{
    DWORD HighLayerInfoType;
    DWORD HighLayerInfoLength;
    UCHAR HighLayerInfo[8];
} ATM_BHLI;

typedef struct
{
    DWORD Layer2Protocol;
    DWORD Layer2UserSpecifiedProtocol;
    DWORD Layer3Protocol;
    DWORD Layer3UserSpecifiedProtocol;
    DWORD Layer3IPI;
    UCHAR SnapID[5];
} ATMJ3LLI;
```

Подробности определения и использования этих полей выходят за рамки нашей книги. Для связи по сети ATM средствами Winsock приложению необходимо присвоить значение *SAP\_FIELD\_ABSENT* следующим полям структур BHLI и BLLI:

*III ATMBIII — Layer2Protocol;*

*III ATMBIII — Layer3Protocol;*

- *ATM BH11 — HighLayerInfoType.*

Когда эти поля получают значение *SAP\_FIELD\_ABSENT*, другие поля обеих структур не используются. Вот пример, иллюстрирующий возможное применение структуры *SOCKADDR\_ATM* для настройки SAP под адрес NSAP:

```
SOCKADDR_ATM atm_addr;
UCHAR MyAddress[ATM_ADDR_SIZE];

atm_addr.satm_family           = AF_ATM;
atm_addr.satm_number.AddressType = ATM_NSAP;
atm_addr.satm_number.NumofDigits = ATM_ADDR_SIZE;
atm_addr.satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
atm_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
atm_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

memcpy(&atm_addr.satm_number.Addr, MyAddress, ATM_ADDR_SIZE);
```

АТМ-адрес, как правило, представляет собой шестнадцатеричную ASCII-строку из 40 символов, которая соответствует 20 байтам, составляющим адрес в стиле NSAP или E 164 в структуре *ATM ADDRESS*. Например, адрес АТМ в стиле NSAP мог бы выглядеть так

47000580FFE1000000F21A1D540000D10FED5800

Преобразование этой строки в 20-байтный адрес может занять довольно много времени, однако Winsock предоставляет независимую от протокола функцию — *WSAStringToAddress*, позволяющую преобразовывать 40-символьную шестнадцатеричную строку ASCII в структуру *ATMADDRESS* (Эта функция подробно описана в конце главы). Другой способ преобразовать шестнадцатеричную строку ASCII в шестнадцатеричный (двоичный) формат — использовать функцию *AtoH* (листинг 6-2). Эта функция не входит в Winsock, но разработать ее просто (см. примеры в главе 7).

Листинг 6-2. Функции, преобразующие шестнадцатеричные строки АТМ

```
//
// Функция AtoH
//
// Описание: Эта функция преобразует АТМ-адрес из строкового
// формата (ASCII) в двоичный (шестнадцатеричный)
//
void AtoH(CHAR *szDest, CHAR «szSource, INT iCount)
{
    while (iCount-)
    {
        *szDest++ = ( BtoH ( *szSource++ ) « 4 )
                + BtoH ( *szSource++ );
    }
    return;
}
>
//
// Функция. BtoH
//
// Описание: Эта функция возвращает эквивалентное двоичное значение
// для отдельного символа в формате ASCII
//
UCHAR BtoH( CHAR ch )
{
    if ( ch >= '0' && ch <= '9' )
    {
        return ( ch - '0' );
    }

    if ( ch >= 'A' && ch <= 'F' )
    {
        return ( ch - 'A' + 0xA );
    }
    , u a> * *.'
```

Листинг 6-2. (продолжение)

```
if (ch >= 'a' && ch <= T )
    return ( ch - 'a' + OA ),
```

// Неверные значения при указании адреса не принимаются

```
return -1;
```

## Создание сокета

В АТМ приложения могут создавать только ориентированные на соединение сокеты, поскольку АТМ позволяет связываться только по VC. Поэтому данные могут быть переданы или как поток байт, или как отдельное сообщение. Для открытия АТМ-сокета вызовите функцию *socket* или *WSASocket* с семейством адресов *AF\_ATM* и типом сокета *SOCK\_RAW*, а полю протокола присвойте *ATMPROTO\_AAL5*—

```
s = socket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5),
s = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, WSA_FLAG_OVERLAPPED),
```

По умолчанию при открытии сокета (как в примере) создается потоковый сокет `ATM Windows` также выводит сведения о поставщике `ATM`, способном передавать данные в виде сообщений. Чтобы его использовать, явно определите поставщик исходного `ATM`-протокола для функции `WSASocket` при помощи структуры `WSAPROTOCOLINFO` (см главу 5). Это необходимо, так как в `Winsock` в запросах `socket` и `WSASOCKET` каждому поставщику `ATM` соответствуют три элемента: семейство адресов, тип сокета и протокол. По умолчанию, `Winsock` возвращает стандартную запись протокола, соответствующую всем трем атрибутам, в нашем случае — это поставщик для потоковой передачи данных. Следующий пример показывает, как найти поставщика `ATM`, передающего данные в виде сообщений, и открыть сокет.

```
dwRet = WSAEnumProtocols(NULL, lpProtocolBuf, idwBufLen),
```

```
for ( i = 0 , i < dwRet, i++)
```

```
if ((lpProtocolBuf[i].iAddressFamily == AF.ATM) &&
    dpProtocolBuf[i].iSocketType == SOCK_RAW) &&
    dpProtocolBuf[i].iProtocol == ATMPROTO_AAL5) &&
    dpProtocolBuf[i].dwServiceFlags &
    XP1JMESSAGE_ORIENTED))
```

```
s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
FROM_PROTOCOL_INFO, lpProtocolBuf[i], 0,
WSA_FLAG_OVERLAPPED);
```

y 1>,.RHM5Rf«M

## Привязка сокета к SAP

Адреса ATM довольно сложны, так как в 20 байтах адреса содержится много информационных элементов. Из них внимания программистов Winsock заслуживает лишь последний байт. Последний байт адреса в стиле NSAP или E.164 соответствует номеру селектора, который уникальным образом позволяет приложению распознать и определить SAP на конечной точке. Как мы уже говорили, Winsock использует SAP для связи по сети ATM.

Приложение-сервер должно зарегистрировать SAP на конечной точке и дожидаться, пока приложение-клиент соединится по зарегистрированному SAP. Для клиента это означает лишь настройку структуры *SOCKADDR\_ATM* с типом *адреса ATM\_E164 или ATM\_NSAP* и предоставление адреса ATM, связанного с SAP сервера. Для создания SAP и ожидания соединений приложение должно сначала создать сокет для семейства адресов *AF\_ATM*. Затем необходимо определить структуру *SOCKADDRATM*, используя такие типы адреса, как *SAP\_FIELD\_ANY\_AESA\_SEL*, *SAP\_FIELD\_ANY\_AESA\_REST*, *ATM\_E164 или ATM\_NoAP* (см. список значений поля *AddressType* в разделе «Протокол ATM Адресация»). Для ATM-сокета SAP создается при вызове приложением функции *bind* (см. главу 7), а типы адреса определяют способ создания SAP на вашей конечной точке.

Тип адреса *SAPFIELD\_ANY\_AESA\_SEL* позволяет создать SAP для прослушивания любого входящего соединения ATM (параметризация адреса ATM и селектора). Это значит, что только один сокет может быть привязан к этой конечной точке для прослушивания любых соединений: при попытке соединения другого сокета с этим типом адреса выдается ошибка *WSAEADDRINUSE*. Впрочем, вы можете явно сопоставить дополнительные сокеты конечной точке для конкретного селектора. Применяя тип адреса *SAP\_FIELD\_ANY\_AESA\_REST*, можно создать SAP, явно привязанную к указанному селектору конечной точки (параметризация адреса ATM без селектора). Нельзя одновременно использовать более одного сокета, привязанного к конкретному селектору на конечной точке, иначе вызов *bind* вернет ошибку *WSAEADDRINUSE*. Для типа *SAP\_FIELD\_ANY\_AESA\_SEL* в структуре *ATM\_ADDRESS* задайте ATM-адрес из нулей, для типа *SAP\_FIELD\_ANY\_AESA\_REST* — обнулите первые 19 байт ATM-адреса. Последний байт должен указывать номер требуемого селектора.

Сокеты, связанные с конкретными селекторами (*SAP\_FIELD\_ANY\_AESA\_REST*), приоритетнее связанных с параметризованными селекторами (*SAP\_FIELD\_ANY\_AESA\_SEL*). А потому первыми при соединениях будут выбраны или они, или сокеты, связанные с явными интерфейсами — *ATM\_NSAP* и *ATME164*. (Если у конечной точки запрошено входящее соединение с указанием явно слушаемого селектора, сокет принимает соединение.) Подключение к сокету с указанием параметризованного селектора произойдет только в отсутствие доступных сокетов, связанных с конкретными селекторами. Создание сокета, прослушивающего соединения по SAP, описано в главе 7.

Наконец, служебная программа *Atmadm.exe* позволяет получить всю информацию об ATM-адресе и виртуальном соединении на конечной точке. Это может пригодиться при разработке приложения ATM, когда понадобится

информация о доступных на конечной точке интерфейсах. Вот параметры командной строки этой программы:

- vc* -с — перечисляет все соединения (VC), удаленный адрес и локальный интерфейс;
- l* -а — перечисляет все зарегистрированные адреса (все локальные интерфейсы ATM и их адреса);
- st* -s — выводит статистику: текущее число запросов, количество принятых и отправленных сигнальных пакетов и ILMI-пакетов и т. п.

## Разрешение имен

В настоящее время для ATM под Winsock нет доступных поставщиков имен. Поэтому, к сожалению, требуется задавать 20-байтный адрес ATM для сокетной связи по сети ATM. В главе 10 рассматривается DNS-пространство Windows 2000, в котором можно регистрировать ATM-адреса с дружественными именами служб.

## Дополнительные функции Winsock 2

Функции *WSAAddressToString* и *WSAStringToAddress* в Winsock 2 обеспечивают независимый от протокола способ преобразования структуры *SOCKADDR* протокола в форматированную строку символов и наоборот. Так как эти функции не зависят от протокола, транспортный протокол должен поддерживать преобразование строк. В настоящее время эти функции работают только для семейств адресов *AF\_INET* и *AF\_ATM*. Функция *WSAAddressToString* определена так:

```
INT WSAAddressToString(
    LPSOCKADDR lpsaAddress,
    DWORD dwAddressLength,
    LPWSAProtocolInfo lpProtocolInfo,
    OUT LPCTSTR lpszAddressString,
    IN OUT LPDWORD lpdwAddressStringLength
);
```

Параметр *lpsaAddress* соответствует структуре *SOCKADDR* для конкретного протокола, содержащего адрес, который надо преобразовать в строку. Параметр *dwAddressLength* задает размер структуры первого параметра для каждого протокола. Необязательный параметр *lpProtocolInfo* представляет поставщик протокола. Поставщиков протокола можно найти функцией *WSAEnumProtocols* (см. главу 5). Если вы зададите *NULL*, вызов использует поставщик первого протокола, поддерживающего семейство адресов из *lpsaAddress*.

Параметр *lpszAddressString* — буфер, где сохраняется удобная для чтения адресная строка. Параметр *lpdwAddressStringLength* — это размер *lpszAddressString*. При выводе в нем возвращается длина строки, фактически скопированной в *lpszAddressString*. Если предоставленный буфер мал, функция выдает ошибку *WSAEFAULT*, а в параметре *lpdwAddressStringLength* вернется требуемый размер в байтах.





## Основы Winsock

Эта глава посвящена изучению основных методик и API-вызовов, необходимых для написания сетевых приложений. Из материалов предыдущей главы вы знаете, как протоколы, доступные из Winsock, адресуют компьютеры и службы. Здесь мы рассмотрим способы установления соединения между двумя компьютерами в сети и механизмы обмена данными. Во избежание повторов, мы обсудим лишь протокол TCP/IP. На прилагаемом компакт-диске содержатся примеры клиент-серверных приложений для каждого из рассмотренных в главе 6 протоколов. Единственная зависящая от протокола операция — это создание сокета. Большинство остальных вызовов функций Winsock, ответственных за установление соединения, отправку и прием данных, не зависят от протокола. Все исключения упоминались в главе 6 при обсуждении конкретных протоколов.

Представленные в этой главе примеры помогут вам лучше понять вызовы Winsock, необходимые для установления соединений и обмена данными. Наша цель — изучить эти вызовы, поэтому в примерах используются прямые блокирующие вызовы Winsock. Другие модели ввода-вывода, реализованные в Winsock, обсуждаются в главе 8.

Кроме того, будут представлены разновидности API-функций для версий Winsock 1 и 2. Если в спецификации Winsock 2 обновлена или добавлена новая API-функция, то ее имя начинается с префикса WSA. Например, имя функции Winsock 1, создающей сокет — *socket*. В Winsock 2 есть ее новая версия — *WSASocket*, использующая расширенные возможности Winsock 2. Исключения из этого правила: функции *WSAStartup*, *WSACleanup*, *WSARecvEx* и *WSAGetLastError*, — упоминаются уже в спецификации Winsock 1.1.

## Инициализация Winsock

Любое Winsock-приложение перед вызовом функции должно загрузить соответствующую версию библиотеки Winsock. Если этого не сделать, функция вернет значение *SOCKET\_ERROR* и выдаст ошибку *WSANOTINITIAUSED*. Загрузку библиотеки Winsock выполняет функция *WSAStartup*:

```
int WSAStartup(  
    WORD wVersionRequested,  
    LPWSADATA lpWSADATA
```

Параметр *wVersionRequested* задает версию загружаемой библиотеки Winsock. Старший и младший байты определяют дополнительный и основной номер версии библиотеки соответственно. Для получения значения параметра *wVersionRequested* можно использовать макрос *MAKEWORD(x, y)*, где *x* — старший байт, *y* — младший.

Параметр *lpWSAData* — указатель на структуру *LPWSADATA*, которая при вызове функции *WSAStartup* заполняется сведениями о версии загружаемой библиотеки:

```
typedef struct WSAData
```

```
WORD          «version;  
WORD          wHighVersion;  
char          szDescription[WSADESCRIPTION_LEN + 1];  
char          szSystemStatus[WSASYS_STATUS_LEN + 1];  
unsigned short iMaxSockets;  
unsigned short iMaxUdpDg;  
char FAR *    lpVendorInfo;
```

```
} WSADATA, FAR * LPWSADATA;
```

*WSAStartup* присваивает параметру *wVersion* значение загружаемой версии. Параметр *wHighVersion* содержит номер последней доступной версии Winsock. Помните, что в обоих полях старший байт определяет дополнительный, а младший — основной номер версии. Поля *szDescription* и *szSystemStatus* заполняются не во всех реализациях Winsock и практически не применяются.

Не используйте и поля *iMaxSockets* и *iMaxUdpDg*. Предполагается, что в них заданы максимальное количество одновременно открытых сокетов и максимальный размер дейтаграммы. Для определения последнего следует запросить сведения о протоколе, вызвав функцию *WSAEnumProtocols*. Максимальное количество одновременно открытых сокетов зависит от свободной физической памяти. Наконец, поле *lpVendorInfo* зарезервировано для информации изготовителя реализации Winsock и не используется ни на одной из платформ Win32.

Разные платформы Windows поддерживают следующие версии Winsock: Windows 95—1.1 (2.2); Windows 98, NT 4.0, 2000 — 2.2; Windows CE — 1.1.

Важно различать основные версии библиотеки. Winsock 1х не поддерживает многие расширенные возможности Winsock, описанные в этом разделе. К тому же, для использования в приложении Winsock 1 необходимо подключить файл Winsock.h, а для Winsock 2 — Winsock2.h.

**ПРИМЕЧАНИЕ** Обновление Winsock 2 для Windows 95 можно найти по адресу <http://www.microsoft.com/windows95/downloads/>.

Даже если платформа поддерживает Winsock 2, не обязательно использовать самую последнюю версию. Напротив, если необходимо, чтобы приложение поддерживалось несколькими платформами, возьмите за основу Winsock 1-й. Такое приложение будет отлично работать на платформе Windows NT 4.0, потому что все вызовы Winsock 1.1 имеются в Winsock 2 DLL.

Как правило, если выходит новая версия Winsock, разработчики стараются ее обновить. В новых версиях исправлены ошибки, к тому же старый код должен без проблем выполняться, по крайней мере, теоретически. В некоторых случаях поведение Winsock отличается от определенного спецификацией. В итоге многие программисты пишут приложения с учетом работы Winsock на конкретной платформе, а не согласно спецификации.

Например, в Windows NT 4.0 при использовании программой модели асинхронных оконных событий после каждого успешного выполнения функции *send* или *WSASend* асинхронно выдается сообщение *FDWRITE*, что указывает на возможность записи данных. Однако в спецификации говорится, что событие *FDWRITE* выдается, когда приложение готово отправлять данные (например, сразу после запуска), и что *FD\_WRITE* означает: следует продолжать запись, пока не будет выдана ошибка *WSAEWOULDBLOCK*. В действительности, после того как система отправит все ожидающие обработки данные и приготовится обрабатывать очередные вызовы *send* и *WSASend*, она отправит окну приложения событие *FDWRITE*-. значит, в этот момент вы можете возобновить запись данных в сеть (статья Q186245 в базе знаний). Эта проблема устранена в четвертом пакете обновлений для Windows NT 4.0 и 2000.

В большинстве случаев при написании новых приложений следует загружать последнюю доступную версию библиотеки Winsock. Если будет выпущена версия 3, приложение, использующее версию 2.2, должно выполняться корректно. При запросе более поздней версии Winsock, не поддерживаемой вашей платформой, *WSAStartup* вернет ошибку, а в поле *wHighVersion* структуры *WSADATA* появится номер последней версии библиотеки, поддерживаемой данной системой.

## Проверка и обработка ошибок

Проверка и обработка ошибок играют весомую роль при написании Winsock-приложения. Функции Winsock достаточно часто возвращают ошибки, но как правило, не критические — передачу информации можно продолжать. Большинство функций Winsock при ошибке вызова возвращают значение *SOCKET\_ERROR*, но так происходит не всегда. При подробном рассмотрении API-вызовов мы обратим внимание на возвращаемые значения, соответствующие ошибкам. Константа *SOCKET\_JSROR* на самом деле равна -1. Для получения более информативного кода ошибки, возникшей после одного из вызовов Winsock, задействуйте функцию *WSAGetLastError*.

```
int WSAGetLastError (void);
```

Эта функция возвращает код последней ошибки. Всем кодам ошибок, возвращаемым *WSAGetLastError*, соответствуют стандартные константные значения. Они описаны в *Winsockh* или в *Winsock2.h* (в зависимости от версии *insock*). Единственное различие этих заголовочных файлов — *Winsock2.h* °Держит больше кодов ошибок новых API-функций. Константы, определенные Для кодов ошибок директивой *\*define*, обычно начинаются с префикса *WSAE*.

## Протоколы с установлением соединения

Сначала мы рассмотрим функции Winsock, необходимые для приема и установления соединений: обсудим, как слушать соединения клиентов, и изучим процесс принятия или отклонения соединения. Затем поговорим о том, как инициировать соединение с сервером. В заключение будет описан процесс передачи данных в ходе сеанса связи.

### Серверные API-функции

Сервер — это процесс, который ожидает подключения клиентов для обслуживания их запросов. Сервер должен прослушивать соединения на стандартном имени. В TCP/IP таким именем является IP-адрес локального интерфейса и номер порта. У каждого протокола своя схема адресации, а потому и свои особенности именования. Первый шаг установления соединения — привязка сокета данного протокола к его стандартному имени функцией *bind*. Второй — перевод сокета в режим прослушивания функцией *listen*. И наконец, сервер должен принять соединение клиента функцией *accept* или *WSAAccept*.

Рассмотрим каждый API-вызов, необходимый для привязки, прослушивания и установления соединения с клиентом. Базовые вызовы, которые клиент и сервер должны сделать для установления канала связи, иллюстрирует рис. 7-1.

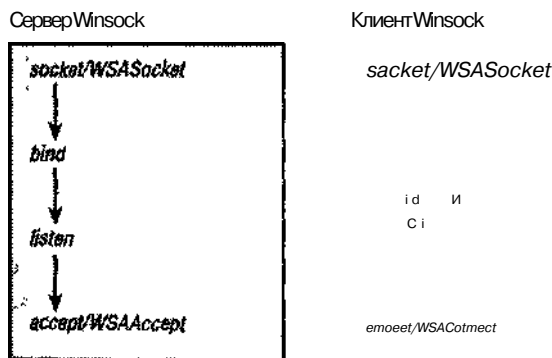


Рис. 7-1. Основные этапы работы клиента и сервера Winsock

#### Функция *bind*

После создания сокета определенного протокола следует связать его со стандартным адресом, вызвав функцию *bind*:

```
int bind(
    SOCKET s,
    const struct sockaddr FAR* name,
    int namelen)
```

Параметр *s* задает сокет, на котором вы ожидаете соединения клиентов. Второй параметр с типом *struct sockaddr* — просто универсальный буфер, фактически, в этот буфер вы должны поместить адрес, соответствующий стандартам используемого протокола, а затем при вызове *bind* привести его к типу *struct sockaddr*. В заголовочном файле Wmsock определен тип *SOCKADDR*, соответствующий структуре *struct sockaddr*. Далее в главе этот тип будет использоваться для краткости. Последний параметр задает размер переданной структуры адреса, зависящей от протокола. Например, следующий код иллюстрирует привязку при TCP-соединении:

```
SOCKET s;
struct sockaddr_in tcpaddr;
int port = 5150;

s = socket(AF_INET, SOCKSTREAM, IPPROTO_TCP);

tcpaddr.sin_family = AF_INET;
tcpaddr.sin_port = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr));
```

Подробнее о структуре *sockaddr\_in* — в разделе, посвященном адресации TCP/IP, главы 6. Там приведен пример создания потокового сокета и последующей настройки структуры адреса TCP/IP для приема соединений клиентов. В данном случае сокет указывает на IP-интерфейс по умолчанию с номером порта 5150. Формально вызов *bind* связывает сокет с IP-интерфейсом и портом.

При возникновении ошибки функция *bind* возвращает значение *SOCKET\_ERROR*. Самая распространенная ошибка при вызове *bind* — *WSAEADDRINUSE*. В случае использования TCP/IP это означает, что с локальным IP-интерфейсом и номером порта уже связан другой процесс, или они находятся в состоянии *TIMEWAIT*. При повторном вызове *bind* для уже связанного сокета возвращается ошибка *WSAEFAULT*.

### Функция *listen*

Теперь нужно перевести сокет в режим прослушивания. Функция *bind* только ассоциирует сокет с заданным адресом. Для перевода сокета в состояние ожидания входящих соединений используется API-функция *listen*.

```
int listen(
    SOCKET s,
    int backlog
```

Первый параметр — связанный сокет. Параметр *backlog* определяет максимальную длину очереди соединений, ожидающих обработки, что важно при запросе нескольких соединений сервера. Пусть значение этого параметра равно 2, тогда при одновременном приеме трех клиентских запросов

первые два соединения будут помещены в очередь ожидания, и приложение сможет их обработать Третий запрос вернет ошибку *WSAECONNREFUSED* После того как сервер примет соединение, запрос удаляется из очереди, а другой — занимает его место Значение *backlog* зависит от поставщика протокола Недопустимое значение заменяется ближайшим разрешенным Стандартного способа получить действительное значение *backlog* нет

Ошибки, связанные с *listen*, довольно просты Самая частая из них — *WSAEINVAL*, обычно означает, что перед *listen* не была вызвана функция *bind* Иногда при вызове *listen* возникает ошибка *WSAEADDRINUSE*, но чаще она происходит при вызове *bind*

## Функции *accept* и *WSAAccept*

Итак, все готово к приему соединений клиентов Теперь вызовем функцию *accept* или *WSAAccept* Прототип *accept*

```
SOCKET accept(
    SOCKET s,
    struct sockaddr FAR* addr,
    int FAR* addrlen
),
```

Параметр *s* — связанный сокет в состоянии прослушивания Второй параметр — адрес действительной структуры *SOCKADDR\_IN*, а *addrlen* — ссылка на длину структуры *SOCKADDR\_IN* Для сокета другого протокола замените *SOCKADDR\_IN* на структуру *SOCKADDR*, соответствующую этому протоколу Вызов *accept* обслуживает первый находящийся в очереди запрос на соединение По его завершении структура *addr* будет содержать сведения об IP-адресе клиента, отправившего запрос, а параметр *addrlen* — размер структуры

Кроме того, *accept* возвращает новый дескриптор сокета, соответствующий принятому клиентскому соединению Для всех последующих операций с этим клиентом должен применяться новый сокет Исходный прослушивающий сокет используется для приема других клиентских соединений и продолжает находиться в режиме прослушивания

В Winsock 2 есть функция *WSAAccept*, способная устанавливать соединения в зависимости от результата вычисления условия

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR * addr,
    LPINT addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
),
```

Первые три параметра — те же, что и в *accept* для Winsock 1 Параметр *lpfnCondition* — указатель на функцию, вызываемую при запросе клиента Она определяет возможность приема соединения и имеет следующий прототип

```

int CALLBACK ConditionFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    (
        LPQOS lpSQOS,
        LPQOS lpGQOS,
        LPWSABUF lpCalleeId,
        LPWSABUF lpCalleeData,
        GROUP FAR * g,
        DWORD dwCallbackData
    ),

```

Передаваемый по значению параметр *lpCallerId* содержит адрес соединяющегося объекта. Структура *WSABUF* используется многими функциями Winsock 2 и определена так

```

typedef struct _WSABUF {
    u_long    len,
    char FAR * buf,
} WSABUF, FAR * LPWSABUF,

```

В зависимости от ее использования, поле *len* определяет размер буфера, на который ссылается поле *buf* или количество данных в буфере *buf*

Для *lpCallerId* параметр *buf* указывает на структуру адреса протокола, по которому осуществляется соединение. Чтобы получить корректный доступ к информации, просто приведите указатель *buf* к соответствующему типу *SOCKADDR*. При использовании протокола TCP/IP это должна быть структура *SOCKADDR\_IN*, содержащая IP-адрес подключающегося клиента. Большинство сетевых протоколов удаленного доступа поддерживают идентификацию абонента на этапе запроса.

Параметр *lpCallerData* содержит данные, отправленные клиентом в ходе запроса соединения. Если эти данные не указаны, он равен *NULL*. Имейте в виду, что большинство сетевых протоколов, таких как TCP/IP, не используют данные о соединении. Чтобы узнать, поддерживает ли протокол эту возможность, обратитесь к соответствующей записи в каталоге Winsock путем вызова функции *WSAEnumProtocols* (см. также главу 5).

Следующие два параметра — *lpSQOS* и *lpGQOS*, задают уровень качества обслуживания, запрашиваемый клиентом. Оба параметра ссылаются на структуру, содержащую сведения о требованиях пропускной способности для приема и передачи. Если клиент не запрашивает параметры качества обслуживания (quality of service, QoS), то они равны *NULL*. Разница между ними в том, что *lpSQoS* используется для единственного соединения, а *lpGQOS* — для групп сокетов. Группы сокетов не реализованы и не поддерживаются в Winsock 1 и 2 (Подробнее о QoS — в главе 12).

Параметр *lpCalleeId* — другая структура *WSABUF*, содержащая локальный адрес, к которому подключен клиент. Снова поле *buf* указывает на объект *SOCKADDR* соответствующего семейства адресов. Эта информация полезна, если сервер запущен на многоадресной машине. Помните, что если сервер связан с адресом *INADDR\_ANY*, запросы соединения будут обслуживаться на Любом сетевом интерфейсе, а параметр — содержать адрес интерфейса, ПРИНЯВШЕГО ГПРЛ1то,...



Параметр *ipCalleeData* дополняет *ipCallerData*. Он ссылается на структуру *WSABUF*, которую сервер может использовать для отправки данных клиенту в ходе установления соединения. Если поставщик услуг поддерживает эту возможность, поле *len* указывает максимальное число отправляемых байт. В этом случае сервер копирует некоторое, не превышающее это значение, количество байт, в блок *buf* структуры *WSABUF* и обновляет поле *len*, чтобы показать, сколько байт передается. Если сервер не должен возвращать данные о соединении, то перед возвращением условная функция приема соединения присвоит полю *len* значение 0. Если поставщик не поддерживает передачу данных о соединении, поле *len* будет равно 0. Опять же, большинство протоколов фактически, все, поддерживаемые платформами Win32 — не поддерживают обмен данными при установлении соединения.

Обработав переданные в условную функцию параметры, сервер должен решить принимать, отклонять или задержать запрос соединения. Если соединение принимается, условная функция вернет значение *CF\_ACCEPT*, если отклоняется — *CF\_REJECT*. Если по каким-либо причинам на данный момент решение не может быть принято, возвращается *CF\_DEFER*.

Как только сервер готов обработать запрос, он вызывает функцию *WSAAccept*. Заметьте, что условная функция выполняется в одном процессе с *WSAAccept* и должна работать как можно быстрее. В протоколах, поддерживаемых платформами Win32, клиентский запрос задерживается, пока не будет вычислено значение условной функции. В большинстве случаев базовый сетевой стек ко времени вызова условной функции уже может принять соединение. А при возвращении значения *CF\_REJECT* стек просто закрывает его. Сейчас мы не будем углубляться в детали использования условной функции принятия соединения — см главу 12.

При возникновении ошибки возвращается значение *INVALID\_SOCKET*, чаще всего — *WSAEWOULDBLOCK*. Оно возникает, если сокет находится в асинхронном или неблокирующем режиме и нет соединения для приема. Если условная функция вернет *CF\_DEFER*, *WSAAccept* генерирует ошибку *WSATRY\_AGAIN*, если *CF\_REJECT-WSAECONNREFUSED*.

## API-функции клиента

Клиентская часть значительно проще и для установления соединения требуется всего три шага: создать сокет функцией *socket* или *WSASocket*, разрешить имя сервера (зависит от используемого протокола), инициировать соединение функцией *connect* или *WSAConnect*.

Из материалов главы 6 вы уже знаете, как создать сокет и разрешить имя IP-узла, так что единственным оставшимся шагом является установление соединения. В главе 6 также рассматривались способы разрешения имен и для других семейств протоколов.

### Состояния TCP

Для работы с Winsock не обязательно знать о состояниях TCP, но с их помощью можно лучше понять, что происходит с протоколом при

вызовах API-функций Winsock. К тому же, многие программисты сталкиваются с одними и теми же проблемами при закрытии сокетов, состояния TCP при этом представляют наибольший интерес.

Начальное состояние любого сокета — CLOSED. Как только клиент инициирует соединение, серверу отправляется пакет SYN, и клиентский сокет переходит в состояние SYN\_SENT. Получив пакет SYN, сервер отправляет пакет SYN-and-ACK, а клиент отвечает на него пакетом ACK. С этого момента клиентский сокет переходит в состояние ESTABLISHED. Если сервер не отправляет пакет SYN-ACK, клиент по истечении времени ожидания возвращается в состояние CLOSED.

Если сокет сервера связан и прослушивает локальный интерфейс и порт, то он находится в состоянии LISTEN. При попытке клиента установить соединение сервер получает пакет SYN и отвечает пакетом SYN-ACK. Состояние сокета сервера меняется на SYN\_RCVD. Наконец, после отправки клиентом пакета ACK сокет сервера переводится в состояние ESTABLISHED.

Существует два способа закрыть соединение. Если этот процесс начинает приложение, то закрытие называется активным, иначе — пассивным. На рис. 7-2 изображены оба вида закрытия. При активном закрытии соединения приложение отправляет пакет FIN. Если приложение вызывает *closesocket win shutdown* (со вторым аргументом *SD\_SEND*), оно отправляет узлу пакет FIN, и состояние сокета меняется на FIN\_WAIT\_1. Обычно узел отвечает пакетом ACK, и сокет переходит в состояние FIN\_WAIT\_2. Если узел тоже закрывает соединение, он отправляет пакет FIN, а компьютер отвечает пакетом ACK и переводит сокет в состояние TIME\_WAIT.

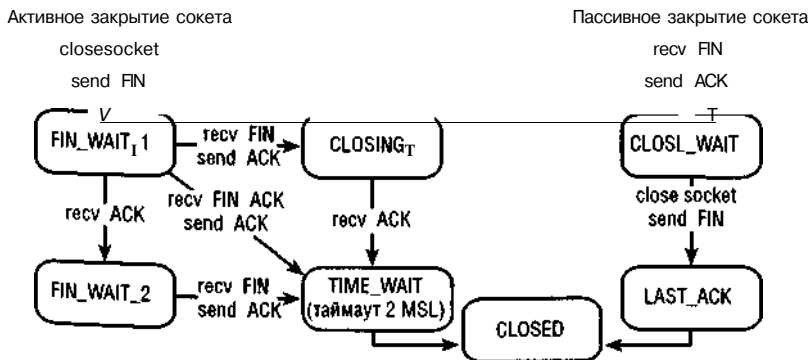


Рис. 7-2. Состояния закрытия сокета TCP

Состояние TIMEWAIT также называется состоянием ожидания MSL. MSL — *максимальное время жизни сегмента* (Maximum Segment lifetime), иными словами, время существования пакета в сети перед его Топорасыванием. У каждого IP-пакета есть поле *времени жизни* (time-live, TTL). Если оно равно 0, значит, пакет можно отбросить. Каждый

маршрутизатор, обслуживающий пакет, уменьшает значение TTL на 1 и передает пакет дальше. Перейдя в состояние TIMEWAIT, приложение остается в нем на протяжении двух периодов времени, равных MSL. Это позволяет TCP в случае потери заключительного пакета ACK послать его заново, с последующей отправкой FIN. По истечении 2MSL сокет переходит в состояние CLOSED.

Результат двух других способов активного закрытия — состояние TIME\_WAIT. В предыдущем случае только одна сторона отправляла FIN и получала ответ ACK, а узел оставался свободным для передачи данных до момента своего закрытия. Здесь и возможны два других способа. В первом случае, при одновременном закрытии, компьютер и узел одновременно запрашивают закрытие: компьютер отправляет узлу пакет FIN и получает от него пакет FIN.

Затем в ответ на пакет FIN компьютер отправляет пакет ACK и изменяет состояние сокета на CLOSING. После получения компьютером пакета ACK от узла сокет переходит в состояние TIME\_WAIT.

Второй случай активного закрытия является вариацией одновременного закрытия: сокет из состояния FIN\_WAIT\_1 сразу переходит в состояние TIMEWAIT. Это происходит, если приложение отправляет пакет FIN и тут же после этого получает от узла пакет FIN-ACK. В таком случае узел подтверждает пакет FIN приложения отправкой своего, на которое приложение отвечает пакетом ACK.

Основной смысл состояния TIMEWAIT заключается в том, что пока соединение ожидает истечения 2MSL, сокетная пара, участвующая в соединении, не может быть использована повторно. Сокетная пара — это комбинация локального и удаленного IP-портов. Некоторые реализации TCP не позволяют повторно использовать любой из портов сокетной пары, находящейся в состоянии TIME\_WAIT. В реализации Microsoft этого дефекта нет. Впрочем, при попытке соединения с сокетной парой, находящейся в состоянии TIMEWAIT, произойдет ошибка *WSAEADDRINUSE*. Одно из решений проблемы (кроме ожидания окончания состояния TIME\_WAIT пары сокетов, использующей локальный порт) — использовать параметр сокета *SO\_REUSEADDR*. Более подробно *SO\_REUSEADDR* рассматривается в главе 9.

И наконец, рассмотрим пассивное закрытие. По этому сценарию приложение получает от узла пакет FIN и отвечает пакетом ACK. В этом случае сокет приложения переходит в состояние CLOSE\_WAIT. Так как узел закрыл свою сторону, он больше не может отправлять данные, но приложение вправе это делать, пока не закроет свою сторону соединения. Для закрытия своей стороны приложение отправляет пакет FIN, после чего TCP-сокет приложения переводится в состояние LAST\_ACK. После получения от узла пакета ACK сокет приложения возвращается в состояние CLOSED.

Подробнее о протоколе TCP/IP — в RFC 793. Этот и другие RFC доступны по адресу <http://wivwrfc-editor.org>

## функции *connect* и *WSAConnect*

Нам осталось обсудить собственно установление соединения. Оно осуществляется вызовом *connect* или *WSAConnect*. Сначала рассмотрим версию Winsock 1 этой функции

```
int connect(
    SOCKET s,
    const struct sockaddr FAR* name,
    int namelen
),
```

Параметры практически не требуют пояснений: *s* — действительный TCP-сокеты для установления соединения, *name* — структура адреса сокета (*SOCKADDRIN*) для TCP, описывающая сервер к которому подключаются, *namelen* — длина переменной *name*. Версия Winsock 2 этой функции определена так

```
int WSAConnect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
),
```

Первые три параметра такие же, как и в функции *connect*. Следующие два *lpCallerData* и *lpCalleeData*, — это строковые буферы, используемые для приема и отправки данных в момент установления соединения. Параметр *lpCallerData* указывает на буфер, содержащий данные, отправляемые клиентом серверу вместе с запросом на соединение, *lpCalleeData* — на буфер с данными, возвращаемыми сервером в ходе установления соединения. Обе переменные являются структурами *WSABUF*, и для *lpCallerData* поле *len* должно указывать длину данных передаваемого буфера *buf*. В случае *lpCalleeData* поле *len* определяет размер буфера *buf* куда принимаются данные от сервера. Два последних параметра *lpSQOS* и *lpGQOS*, — ссылаются на структуры QoS, определяющие требования пропускной способности отправки и приема данных устанавливаемого соединения. Параметр *lpSQOS* указывает требования к сокету *s*, а *lpGQOS* — к группе сокетов. На данный момент группы сокетов не поддерживаются. Нулевое значение *lpSQOS* означает, что приложение не предъявляет требований к качеству обслуживания.

Если на компьютере, к которому вы подключаетесь, не запущен процесс, прослушивающий данный порт, функция *connect* вернет ошибку *WSAECONNREFUSED*. Другая ошибка — *WSAETIMEDOUT*, происходит, когда вызываемый адрес недоступен, например, из-за отказа коммуникационного оборудования на пути к узлу или отсутствия узла в сети.

## Передача данных

По сути, в сетевом программировании самое главное — уметь отправлять и принимать данные. Для пересылки данных по сокету используются функции *send* и *WSASend*. Аналогично, для приема данных существуют функции *recv* и *WSARecv*.

Заметьте, что все буферы, используемые при отправке и приеме данных состоят из элементов типа *char*. То есть эти функции не предназначены для работы с кодировкой UNICODE. Это особенно важно для Windows CE, так как она использует UNICODE по умолчанию. Отправить строку символов UNICODE можно двумя способами: в исходном виде или привести к типу *char* '. Тонкость в том, что при указании количества отправляемых или принимаемых символов результат функции, определяющей длину строки, нужно умножить на 2, так как каждый UNICODE-символ занимает 2 байта строкового массива. Другой способ: сначала перевести строку из UNICODE в ASCII функцией *WideCharToMultiByte*.

Все функции приема и отправки данных при возникновении ошибки возвращают код *SOCKETJERROR*. Для получения более подробной информации об ошибке вызовите функцию *WSAGetLastError*. Самые распространенные ошибки - *WSAECONNABORTED* и *WSAECONNRESET*. Обе возникают при закрытии соединения: либо по истечении времени ожидания, либо при закрытии соединения партнерским узлом. Еще одна типичная ошибка — *WSAEWOULDBLOCK*, обычно происходит при использовании неблокирующих или асинхронных сокетов. По существу, она означает, что функция не может быть выполнена в данный момент. В главе 8 будут описаны разные методы ввода-вывода Winsock, которые помогут избежать этих ошибок.

## Функции *send* и *WSASend*

API-функция *send* для отправки данных по сокету определена так:

```
int send(
    SOCKET s,                                //
    const char FAR * buf,                    // if
    int len,                                  //
    int flags                                 //
);
```

Параметр *s* определяет сокет для отправки данных. Второй параметр — *buf*, указывает на символьный буфер, содержащий данные для отправки. Третий — *len*, задает число отправляемых из буфера символов. И последний параметр — *flags*, может принимать значения 0, *MSGDONTROUTE*, *MSG\_OOB*, или результат логического ИЛИ над любыми из этих параметров. При указании флага *MSGDONTROUTE* транспорт не будет маршрутизировать отправляемые пакеты. Обработка этого запроса остается на усмотрение базового протокола (например, если транспорт не поддерживает этот параметр", запрос игнорируется). Флаг *MSG\_OOB* указывает, что данные должны быть отправлены *вне полосы* (out of band), то есть срочно.

При успешном выполнении функция *send* вернет количество переданных байт, иначе — ошибку *SOCKET\_ERROR*. Одна из типичных ошибок — *WSAENOTABORTED*, происходит при разрыве виртуального соединения из-за ошибки протокола или истечения времени ожидания. В этом случае сокет <sup>же</sup> <sup>н</sup> быть закрыт, так как он больше не может использоваться. Ошибка *WSAENOTABORTED* происходит, если приложение на удаленном узле, выполнив аппаратное закрытие, сбрасывает виртуальное соединение, или неожиданно завершается, или происходит перезагрузка удаленного узла. В этой ситуации сокет также должен быть закрыт. Еще одна ошибка — *WSAETIMEDOUT*, зачастую происходит при обрыве соединения по причине сбоя сети или отказа удаленной системы без предупреждения.

функция Winsock версии 2 *WSASend* — аналог *send*, определена так:

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Сокет является действительным описателем сеанса соединения. Второй параметр указывает на структуру *WSABUF* или на массив этих структур. Третий — определяет число передаваемых структур *WSABUF*. Помните, что структура *WSABUF* включает сам символьный буфер и его длину. Может возникнуть вопрос: зачем нужно отправлять более одного буфера за раз? Это называется *комплексным вводом-выводом* (scatter-gather I/O). Подробнее мы обсудим его далее, а сейчас лишь отметим, что при использовании нескольких буферов для отправки данных по сокету соединения массив буферов отправляется, начиная с первой и заканчивая последней структурой *WSABUF*.

Параметр *lpNumberOfBytesSent* — указатель на тип *DWORD*, который после вызова *WSASend* содержит общее число переданных байт. Параметр флагов *dwFlags* такой же, что и в функции *send*. Последние два указателя — *lpOverlapped* и *lpCompletionRoutine* используются для *перекрываемого ввода-вывода* (overlapped I/O) — одной из моделей асинхронного ввода-вывода, поддерживаемых Winsock (см. также главу 8).

*WSASend* присваивает параметру *lpNumberOfBytesSent* количество записанных байт. При успешном выполнении функция возвращает 0, иначе — *SOCKET\_ERROR*. Ошибки те же, что и у функции *send*.

## Функция *WSASendDisconnect*

то специализированная функция используется редко. Она определена так:

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOut boundDisconnectData
    i--w
```

## Срочные данные

Если приложению требуется отправить через потоковый сокет информацию более высокого приоритета, оно может обозначить эти сведения как *срочные данные* (out-of-band, OOB). Приложение с другой стороны соединения получает и обрабатывает OOB-данные через отдельный логический канал, концептуально независимый от потока данных.

В TCP передача OOB-данных реализована путем добавления 1-битового маркера (называемого URG) и 16-битного указателя в заголовке сегмента TCP, которые позволяют выделить важные байты в основном трафике. На данный момент для TCP существуют два способа выделения срочных данных. В RFC 793, описывающем TCP и концепцию срочных данных, говорится, что указатель срочности в заголовке TCP является положительным смещением байта, следующего за байтом срочных данных. Однако в RFC 1122 это смещение трактуется, как указатель на сам байт срочности.

В спецификации Winsock под термином OOB понимают как независимые от протокола OOB-данные, так и реализацию механизма передачи срочных данных в TCP. Для проверки, есть ли в очереди срочные данные, вызовите функцию *ioctlsocket* с параметром *SIOCATMARK*. Подробнее об этой функции — в главе 9.

В Winsock предусмотрено несколько способов передачи срочных данных. Можно встроить их в обычный поток, либо, отключив эту возможность, вызвать отдельную функцию, возвращающую только срочные данные. Параметр *SOJDOBINLJNE* управляет поведением OOB-данных (он подробно обсуждается в главе 9).

В ряде случаев срочные данные используют программы Telnet и Rlogin. Впрочем, если вы не планируете писать собственные версии этих программ, избегайте применения срочных данных — они не стандартизированы и могут иметь другие реализации на отличных от Win32 платформах. Если вам нужно время от времени передавать срочно какую-то информацию, создайте отдельный управляющий сокет для срочных данных, а основное соединение предоставьте для обычной передачи данных.

Функция *WSASendDisconnect* начинает процесс закрытия сокета и отправляет соответствующие данные. Разумеется, она доступна только для протоколов, поддерживающих постепенное закрытие и передачу данных при его осуществлении. Ни один из существующих поставщиков транспорта на данный момент не поддерживает передачу данных о закрытии соединения. Функция *WSASendDisconnect* действует аналогично *shutdown* с параметром *SD\_SEND*, но также отправляет данные, содержащиеся в параметре *boundDisconnectData*. После ее вызова отправлять данные через сокет невозможно. В случае неудачного завершения *WSASendDisconnect* возвращает значение *SOCKET\_ERROR*. Ошибки, встречающиеся при работе функции, аналогичны ошибкам *send*.

## функции *recv* и *WSARecv*

Функция *recv* — основной инструмент приема данных по сокету. Она определена так:

```
int recv(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags
);
```

Параметр *s* определяет сокет для приема данных. Второй параметр — *buf* является символьным буфером и предназначен для полученных данных, а *len* указывает число принимаемых байт или размер буфера *buf*. Последний параметр — *flags*, может принимать значения 0, *MSG\_PEEK*, *MSG\_WAITFORONE* или результат логического ИЛИ над любыми из этих параметров. Разумеется, 0 означает отсутствие особых действий. Флаг *MSG\_PEEK* указывает, что доступные данные должны копироваться в принимающий буфер и при этом оставаться в системном буфере. По завершении функция также возвращает количество ожидающих байт.

Считывать сообщения таким образом не рекомендуется. Мало того, что из-за двух системных вызовов (одно! — для считывания данных, и другого, без флага *MSG\_PEEK* — для удаления данных), снижается производительность. В ряде случаев этот способ просто не надежен. Объем возвращаемых данных может не соответствовать их суммарному доступному количеству. К тому же, сохраняя данные в системных буферах, система оставляет все меньше памяти для размещения входящих данных. В результате уменьшается размер окна TCP для всех отправителей, что не позволяет приложению достичь максимальной производительности. Лучше всего скопировать все данные в собственный буфер и обрабатывать их там. Флаг *MSG\_WAITFORONE* уже обсуждался ранее при рассмотрении отправки данных.

Использование *recv* в сокетах, ориентированных на передачу сообщений или дейтаграмм, имеет несколько особенностей. Если при вызове *recv* размер ожидающих обработки данных больше предоставляемого буфера, то после его полного заполнения возникает ошибка *WSAEMSGSIZE*. Заметьте: ошибка превышения размера сообщения происходит только при использовании протоколов, ориентированных на передачу сообщений. Поточковые протоколы буферизируют поступающие данные и при запросе приложением предоставляют их в полном объеме, даже если количество ожидающих обработки данных больше размера буфера. Таким образом, ошибка *WSAEMSGSIZE* не может произойти при работе с потоковыми протоколами.

Функция *WSARecv* обладает дополнительными по сравнению с *recv* возможностями: поддерживает перекрытый ввод-вывод и фрагментарные действия. Таграммные уведомления

```
int *WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
```



```

    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);

```

Параметр 5 — сокет соединения. Второй и третий параметры определяют буферы для приема данных. Указатель *lpBuffers* ссылается на массив структур *WSABUF*, а *dwBufferCount* — определяет количество таких структур в массиве. Параметр *lpNumberOfBytesReceived* в случае немедленного завершения операции получения данных указывает на количество принятых этим вызовом байт. Параметр *lpFlags* может принимать значения *MSG\_PEEK*, *MSG\_WAITFORONE*, *MSG\_PARTIAL* или результат логического ИЛИ над любыми из этих параметров.

У флага *MSG\_PARTIAL* в зависимости от способа использования могут быть разные значения и смысл. Для протоколов, ориентированных на передачу сообщений, этот флаг задается после вызова *WSARecv* (если все сообщение не может быть возвращено из-за нехватки места в буфере). В этом случае каждый последующий вызов *WSARecv* задает флаг *MSG\_PARTIAL*, пока сообщение не будет прочитано целиком. Если этот флаг передается как входной параметр, операция приема данных должна завершиться, как только данные будут доступны, даже если это только часть сообщения. Флаг *MSG\_PARTIAL* используется только с протоколами, ориентированными на передачу сообщений. Запись каждого протокола в каталоге Winsock содержит флаг, указывающий на поддержку этой возможности (см. также главу 5). Параметры *lpOverlapped* и *lpCompletionROUTINE* применяются в операциях перекрытого ввода-вывода (обсуждаются в главе 8).

## Функция *WSARecvDisconnect*

Эта функция обратна *WSASendDisconnect* и определена так:

```

int WSARecvDisconnect(
    SOCKET s,
    LPWSABUF lpInboundDisconnectData
);

```

Как и у *WSASendDisconnect*, ее параметрами являются описатель сокета соединения и действительная структура *WSABUF* для приема данных. Функция принимает только данные о закрытии соединения, отправленные с другой стороны функцией *WSASendDisconnect*, ее нельзя использовать для приема обычных данных. К тому же, сразу после принятия данных она прекращает прием с удаленной стороны, что эквивалентно вызову *shutdown* с параметром *SD\_RECV*.

## Функция *WSARecvEx*

Эта функция — специальное расширение Microsoft для Winsock 1. Она идентична *recv* во всем, кроме того, что параметру *flags* передается по ссылке. <sup>\*\*\*</sup> позволяет базовому поставщику задавать флаг *MSG\_PARTIAL*.

```
int PASCAL FAR WSARcvEx(
    SOCKET s,
    char FAR « buf,
    int len,
    int «flags
);
```

Если полученные данные не составляют полного сообщения, в параметре *flags* возвращается флаг *MSG\_PARTIAL*. Он используется только с протоколами, ориентированными на передачу сообщений. Когда при принятии неполного сообщения флаг *MSG\_PARTIAL* передается как часть параметрами<sup>е</sup>, функция завершается немедленно, вернув принятые данные. Если в буфере не хватает места, чтобы принять сообщение целиком, *WSARcvEx* вернет ошибку *WSAEMSGSIZE*, а оставшиеся данные будут отброшены. Обратите внимание на разницу между флагом *MSG\_PARTIAL* и ошибкой *WSAEMSGSIZE*: в случае ошибки сообщение поступило целиком, однако соответствующий буфер слишком мал для его приема. Флаги *MSG\_PEEK* и *MSG\_DONTWAIT* также можно использовать в *WSARcvEx*.

## Потоковые протоколы

Большинство протоколов с установлением соединения являются потоковыми. Важно учитывать, что при использовании любой функции отправки или приема данных через потоковый сокет нет гарантии, что вы прочитаете или запишете весь запрошенный объем данных. Скажем, требуется отправить 2048 байт из символьного буфера функцией *send*:

```
char sendbuff[2048];
int nBytes = 2048;
```

```
// Заполнение буфера sendbuff 2048 байтами данных
```

```
// Присвоение s значения действительного потокового сокета соединения
ret = send(s, sendbuff, nBytes, 0);
```

Возможно, функция *send* сообщит об отправке менее 2048 байт. Переменная *ret* будет содержать количество переданных байт, поскольку система выделяет определенное количество буферного пространства на отправку и прием сообщений для каждого сокета. При отправке данных внутренние буферы удерживают их до момента отправки непосредственно по проводу, фичиной неполной отправки может быть, например, передача большого количества данных, при этом все буферы слишком быстро заполняются.

В TCP/IP также существует так называемый размер окна. Принимающая сторона регулирует его, указывая количество данных, которое способна принять. При переполнении данными получатель может задать нулевой размер <sup>кна</sup> чтобы справиться с поступившими данными. Это приведет к приостановке отправки данных, пока размер окна не станет больше 0. В нашем слу<sup>е</sup> <sup>аз</sup> размер буфера может оказаться равным 1024 байтам, следовательно, по<sup>е</sup> Р<sup>аз</sup> уется повторно отправить оставшиеся 1024 байта. Отправку всего содерж<sup>а</sup>го буфера обеспечит следующий фрагмент программы:

```

char sendbuff[2048];
int  nBytes = 2048,
     nLeft,
     idx;

// Заполнение буфера sendbuff 2048 байтами данных

// Присвоение s значения действительного потокового сокета соединения
nLeft = nBytes;
idx = 0;
while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Ошибка
        nLeft -= ret;
        idx += ret;
    }
}

```

Все, сказанное далее справедливо и для приема данных на потоковом сокете, но для нас это не очень важно. Приложение не знает, сколько данных оно в очередной раз прочтет на потоковом сокете. Если вам нужно отправить дискретные сообщения по потоковому протоколу, это не составит труда. Например, если у всех сообщений одинаковый размер (512 байт), то прочитать их можно так:

```

char  recvbuff[1024];
int   ret,
     nLeft,
     idx;

nLeft = 512;
idx = 0;
while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Ошибка
    }
    idx += ret;
    nLeft -= ret;
}

```

Ситуация несколько усложнится, если размер сообщений будет варьироваться. Тогда потребуется реализовать собственный протокол, сообщающий получателю о размере поступающего сообщения. Пусть первые 4 байта сообщения указывают его размер в виде целого числа. Преобразовав их в число при чтении, приемник узнает длину сообщения.

### Комплексный ввод-вывод

Впервые принцип *комплексного ввода-вывода* (Scatter-Gather I/O) был применен в функциях *recv* и *wrtev* сокетов Беркли (Berkeley Sockets). В Winsock 2 его поддерживают функции *WSARecv*, *WSARecvFrom*, *WS - Send* и *WSASenaTo*. Комплексный ввод-вывод наиболее полезен для п; i-ложений, отправляющих и принимающих данные в специфическом формате. Например, если передаваемые клиентом серверу сообщения должны состоять из фиксированного 32-байтного заголовка, определяющего некие действия, 64-байтного блока данных и заканчиваться 16-байтной контрольной суммой. В этом случае функция *WSASend* может быть вызвана для массива соответствующих трех структур *WSABUF*. На принимающей стороне одним из входных параметров вызываемой функции *WSARecv* также должны быть три структуры *WSABUF*, содержащие 32, 64 и 16 байт.

При использовании потоковых сокетов операции комплексного ввода-вывода просто интерпретируют несколько буферов данных как один непрерывный. Функция принятия данных может завершиться раньше, чем будут заполнены все буферы. В сокетах, ориентированных на передачу сообщений, каждая операция получения данных принимает одно сообщение, длина которого не больше размера буфера. Если в буфере недостаточно места, вызов заканчивается ошибкой *WSAE-MSGSIZE*, и данные усекаются до размера доступного пространства. Разумеется, в протоколах, поддерживающих фрагментарные сообщения, для предотвращения потери данных можно использовать флаг *MSGPARTIAL*.

## Завершение сеанса

По окончании работы с сокетом необходимо закрыть соединение и освободить все ресурсы, связанные с описателем сокета, вызвав функцию *dosesocket*. Впрочем, ее неправильное использование может привести к потере данных. Поэтому перед вызовом *dosesocket* сеанс нужно корректно завершить функцией *shutdown*.

### Функция *shutdown*

Правильно написанное приложение уведомляет получателя об окончании отправки данных. Так же должен поступить и узел. Такое поведение называется корректным завершением сеанса и осуществляется с помощью функции *shutdown*-.

```
int shutdown(  
    SOCKET s,  
    int how
```

Параметр *how* может принимать значения *SD\_RECEIVE*, *SD\_SEND* или *SD\_BOTH*. Значение *SD\_RECEIVE* запрещает все последующие вызовы любых

функций приема данных, на протоколы нижнего уровня это не действует. Если в очереди TCP-сокета есть данные, либо они поступают позже, соединение сбрасывается. UDP-сокеты в аналогичной ситуации продолжают принимать данные и ставить их в очередь. *SDJSEND* запрещает все последующие вызовы функций отправки данных. В случае TCP-сокетов после подтверждения получателем приема всех отправленных данных передается пакет *FIN*. Наконец, *SDBOTH* запрещает как прием, так и отправку.

### Функция *closesocket*

Эта функция закрывает сокет. Она определена так:

```
int closesocket (SOCKET s);
```

Вызов *closesocket* освобождает дескриптор сокета, и все дальнейшие операции с сокетом закончатся ошибкой *WSAENOTSOCK*. Если не существует других ссылок на сокет, все связанные с дескриптором ресурсы будут освобождены, включая данные в очереди.

Ожидающие асинхронные вызовы, исходящие от любого потока данного процесса, отменяются без уведомления. Ожидающие операции перекрытого ввода-вывода также аннулируются. Все выполняющиеся события, процедура и порт завершения, связанные с перекрытым вводом-выводом, завершатся ошибкой *WSA\_OPERATION\_ABORTED*. (Асинхронные и неблокирующие модели ввода-вывода более подробно обсуждаются в главе 8.) Другой фактор, влияющий на поведение функции *closesocket*, — значение параметра сокета *SOJLINGER* (его полное описание — в главе 9).

### Пример

Разнообразие функций отправки и приема данных удивительно. Но в действительности большинству приложений для приема информации требуются только *recv* или *WSARecv*, а для отправки — *send* или *WSASend*. Другие функции обеспечивают специальные возможности и редко используются (или поддерживаются только транспортными протоколами).

Теперь рассмотрим небольшой пример клиент-серверного взаимодействия с учетом описанных принципов и функций. В листинге 7-1 приведен код простого эхо-сервера. Приложение создает сокет, привязывает его к локальному IP-интерфейсу и порту, и слушает соединения клиентов. После приема от клиента запроса на соединение создается новый сокет, который передается клиентскому потоку. Поток читает данные и возвращает их клиенту.

#### Листинг 7-1. Эхо-сервер

```
// Имя модуля: Server.c

// Описание:
// Это пример простого сервера TCP, принимающего
// соединения клиентов. После установления соединения
// порождается процесс, который получает данные клиента
// и отправляет их обратно (если параметр возврата данных
// не выключен).
```

## Листинг 7-1. (продолжение)

//

// Параметры компиляции:

// cl -o Server Server.c ws2\_32.lib

// Параметры командной строки:

// server [-p:x] [-i:IP] [-o]

// -p:x Номер порта, на котором будут прослушиваться соединения

// -i:str Интерфейс, на котором будут прослушиваться соединения

// -o Только принимать данные, не возвращая их клиенту

//

«include <wssock2.h>

«include <stdio.h>

«include <stdlib.h>

\*\*

«define DEFAULT\_PORT 5150

«define DEFAULT\_BUFFER 4096

int iPort = DEFAULT\_PORT; // Порт прослушивания клиентов

BOOL bInterface = FALSE, // Прослушивать указанный интерфейс

bRecvOnly = FALSE; // Только прием данных

char szAddress[128]; // Интерфейс прослушивания клиентов

// Функция: usage

//

// Описание:

// Выводит сведения о параметрах командной строки и выходит

//

void usage()

{

printf("usage: server [-p:x] [-i:IP] [-o]\n\n");

printf(" -p:x Port number to listen on\n");

printf(" -i:str Interface to listen on\n");

printf(" -o Don't echo the data back\n\n");

ExitProcess(i);

// «Функция: ValidateArgs

//

// Описание:

// Анализирует параметры командной строки и задает

/ некоторые глобальные флаги для указания выполняемых действий

\*ValidArgs(int argc, char \*\*argv)

{

int i; for (i = 1; i < argc; i++)

if (argv[i][0] == '-')

switch (argv[i][1])

{

ЛИСТИНГ 7-1. (продолжение)

```
for(i = 1; i < argc;
    if ((argv[i][0] == '-') || (argv[i][0] == '/'))
    {
        switch (tolower(argv[i][1]))
        {
            case 'p':
                iPort = atoi(&argv[i][3]);
                break;
            case 'i':
                blInterface = TRUE;
                if (strlen(argv[i]) > 3)
                    strcpy(szAddress, &argv[i][3]);
                break;
            case 'o':
                bRecvOnly = TRUE;
                break;
            default:
                usage();
                break;
        }
    }
}
```

```
// Функция: ClientThread
```

// ● \*

## II Описание:

```
// Вызывается в качестве потока, управляет данным клиентским
// соединением. Входным параметром является описатель сокета,
// возвращаемый функцией accept(). Функция получает данные
// от клиента и отправляет их обратно.
```

**DWORD WINAPI ClientThread(LPVOID lpParam)**

```
SOCKET      sock=(SOCKET)lpParam;  
char        szBuff[DEFAULT_BUFFER];  
int         ret,  
            nLeft,  
            idx;
```

> whiled)                    те . N, Лонднак  
' {                         Г Г.- V-

```
// Блокирующий вызов resv()
```

```
// -π» sat)
```

```
ret = recv(sock, szBuff, DEFAULT.BUFFER, 0);
```

```
(ret == 0) // Корректное завершение ^
break;
```

## Листинг 7-1. (продолжение)

```

else if (ret == SOCKET.ERROR)
{
    printf("recv() failed: Xd\n", WSAGetLastError());
    break;
}
szBuff[ret] = '\0';
printf("RCV: 'Xs'\n", szBuff);
//
// Возврат данных клиенту, если задан соответствующий параметр
//
if (IbRecvOnly)
{
    nLeft = ret;
    idx = 0;
    i
    -, nfh

I II Проверка, что все данные записаныдар
//
while(nLeft > 0)
{
    ret = send(sock, &szBuff[idx], nLeft, 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET.ERROR)
    {
        printf("send() failed: Xd\n",
            WSAGetLastError());
        break;
    }
    nLeft -= ret;
    idx += ret;
}

return 0;

// Функция: main4 * *x
'I ' i w
II Описание:
// Главный поток выполнения. Инициализирует Winsock, анализирует
// параметры командной строки, создает и прослушивает сокет,
/ привязывает его к локальному адресу и ждет подключений клиентов

jnt maindnt argc, char *.argv)

WSADATA wsd;
SOCKET sListen,

```

см. след. стр.

hit



Листинг 7-1. *(продолжение)*

```

        sClient;
    int      iAddrSize;
    HANDLE    hThread;
    DWORD     dwThreadId;
    struct sockaddr_in local,
                client;

    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    <
        printf("Failed to load Winsock!\n");
        return 1;
    }
    // Создание сокета прослушивания
    //
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sListen == SOCKET_ERROR)
    {
        printf("socket() failed: Jfd\n", WSAGetLastError0);
        return 1;
    }
    // Выбор локального интерфейса и привязка к нему
    , //
    if (blInterface)

//      local.sin_addr.s_addr = inet_addr(szAddress);
    if (local.sin_addr.s_addr == INADDR_NONE)
        usage();
7
•7 else
    local.sin_addr.s_addr = htonl(INADDR_ANY);
7, local.sin_family = AF_INET;
    local.sin_port = htons(iPort);
>Ж if (bind(sListen, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: Xd\n", WSAGetLastError0);
        return 1;

    listen(sListen, 8); >

    // Ожидание клиентов в бесконечном цикле.
    // Создание потока в случае обнаружения и передача ему описателя.
    //
    while (1)
    {
        iAddrSize = sizeof(client);
        'sClient = accept(sListen, (struct sockaddr *)&client,
            &iAddrSize);

```

**Листинг 7-1. {продолжение}**

```

    if (sClient == INVALID_SOCKET)
    {
        pnntf("accept() failed: Xd\n", WSAGetLastErrorO);
        break;
    }
    printf("Accepted client: Xs:Xd\n",
        inet_ntoa(client.sin_addr), ntohs(client.sin_port));

    hThread = CreateThread(NULL, 0, ClientThread,
        (LPVOID)sClient, 0, idwThreadId);
    if (hThread == NULL)
    {
        pnntf("CreateThread() failed: Xd\n", GetLastError());
        break;
    }
    >
    CloseHandle(hThread);
}
closesocket(sListen);

WSACleanupO;
return 0;
}

```

Клиентская часть данного примера (листинг 7-2) значительно проще. Клиент создает сокет, разрешает переданное приложению имя сервера и соединяется с сервером. Как только соединение установлено, серверу отправляется несколько сообщений. После каждой отправки клиент ожидает эхо-ответа сервера. Все полученные по сокету данные выводятся на экран.

Эхо-взаимодействие клиента и сервера не вполне отражает потоковую сущность TCP — для клиента каждая операция чтения сопровождается операцией записи, а для сервера наоборот. Таким образом, каждый вызов сервером функции чтения почти всегда возвращает все сообщение, отправленное клиентом. Почти, но не всегда — если размер сообщения клиента превышает максимальную единицу передачи для TCP, то оно разбивается на отдельные пакеты и получателю потребуется несколько раз вызвать функцию приема данных. Для лучшей иллюстрации потоковой передачи запустите клиент и сервер с параметром -o. Тогда клиент будет только отправлять данные, а сервер — только принимать:

```
server -p:5150 -o
```

```
client -p:5150 -s:IP -n:10 -o
```

Корее всего, вы увидите, **что клиент совершает десять отправок, а сервер получает все десять сообщений за один или два вызова recv.**

**Листинг 7-2. Эхо-клиент**

```
//Имя модуля; Client.c
```

см. след. стр.



**Листинг 7-2.** {продолжение}

Ц описание:

// Анализирует параметры командной строки и задает  
// некоторые глобальные флаги для указания выполняемых действий

void ValidateArgsdnt argc, char \*\*argv)

```

int          il          <it*

for(i = 1; i < argc; i++)  ЩфТМ
{
    if ((argv[i][0] == '-') || (argv[i][0] == '/'))
    {
        switch (tolower(argv[i][1]))
        {
            case 'p':          // Удаленный порт
                if (strlen(argv[i]) > 3)
                {
                    iPort = atoi(&argv[i][3]);
                    break;
                }
            case 's':          // Сервер
                if (strlen(argv[i]) > 3)
                {
                    strcpy(szServer, &argv[i][3]);
                    break;
                }
            case 'n':          // Число отправок сообщения
                if (strlen(argv[i]) > 3)
                {
                    dwCount = atol(&argv[i][3]);
                    break;
                }
            case 'o':          // Только отправка сообщений, без приема
                bSendOnly = TRUE;
                break;
            default:
                usage();      /*qo| те^втугеегоо »н
                break;      N ,»ав¥ *>« ore

                                (З«0й Я'ШАЙ! »• ibbti

```

// Функция: main

//

// Описание:

- I Главный поток выполнения. Инициализирует Winsock, анализирует параметры командной строки, создает сокет, соединяется с сервером,
- / отправляет и принимает данные.

j UM argc, char ..argv)

w s d ;

см. след. стр.

Листинг 7-2. (продолжение)

```

sock_t sClient;
char szBuffer[DEFAULT_BUFFER];
int ret;
i;

struct sockaddr_in server;
struct hostent *host = NULL;

// Анализ командной строки и загрузка Wmsock
{
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Wmsock library\n");
        return 1;
    }

    strcpy(szMessage, DEFAULT_MESSAGE);

    //
    // Создание сокета и попытка подключения к серверу
    //
    sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sClient == INVALID_SOCKET)
    {
        printf("socket() failed: %d\n", WSAGetLastError());
        return 1;
    }

    server.sin_family = AF_INET;
    server.sin_port = htons(iPort);
    server.sin_addr.s_addr = inet_addr(szServer);

    //
    // Если адрес сервера не соответствует форме
    // "aaa.bbb.ccc.ddd", значит это имя узла, и его следует разрешить
    //
    if (server.sin_addr.s_addr == INADDR_NONE)
    {
        host = gethostbyname(szServer);
        if (host == NULL)
        {
            printf("Unable to resolve server: %s\n", szServer);
            return 1;
        }
        CopyMemory(&server.sin_addr, host->h_addr_list[0],
            host->h_length);
    }

    if (connect(sClient, (struct sockaddr *)server,
        sizeof(server)) == SOCKET_ERROR)
    {
        printf("connect() failed: %d\n", WSAGetLastError());
        return 1;
    }
}

```

**Листинг 7-2.** (продолжение)

II Отправка и прием данных

```

for(i = 0; i < dwCount;

    ret = send(sClient, szMessage, strlen(szMessage), 0);
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        printf("send() failed: Xd\n", WSAGetLastError());
        break;
    }

    printf("Send Xd bytes\n", ret);
    if (IbSendOnly)
    {
        ret = recv(sClient, szBuffer, DEFAULT_BUFFER, 0);
        if (ret == 0)           // Корректное завершение сеанса
            break;
        else if (ret == SOCKET_ERROR)
        {
            printf("recv() failed: Xd\n", WSAGetLastError());
            break;
        }

        szBuffer[ret] = '\0';
        printf("RECV [Xd bytes]: 'Xs'\n", ret, szBuffer);

        closesocket(sClient);
    }
    WSACleanup();
    return 0;
}

```

## Протоколы, не требующие соединения

финцип действия таких протоколов иной, так как в них используются другие методы отправки и приема данных. Обсудим сначала получателя (или сервер), потому что не требующий соединения приемник мало отличается от серверов, требующих соединения.

### Приемник

Процесс получения данных на сокете, не требующем соединения, прост.

Начала создают сокет функцией *socket* или *WSASocket*. Затем выполняют привязку сокета к интерфейсу, на котором будут принимать данные, функцией *bind* (как и в случае протоколов, ориентированных на сеансы). Разница<sup>а</sup> в том, что нельзя вызвать *listen* или *accept*: вместо этого нужно просто

ожидать приема входящих данных. Поскольку в этом случае соединения нет, принимающий сокет может получать дейтаграммы от любой машины в сети. Простейшая функция приема — *recvfrom*—.

```
int recvfrom(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

Первые четыре параметра такие же, как и для функции *recv*, включают допустимые значения для *flags*— *MSG\_DONTWAIT* и *MSG\_PEEK*. Параметр *from* — структура *SOCKADDR* для данного протокола слушающего сокета, на размер структуры адреса ссылается *fromlen*. После возврата вызова структура *SOCKADDR* будет содержать адрес рабочей станции, которая отправляет данные.

В Winsock 2 применяется другая версия *recvfrom* — *WSARecvFrom*—.

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Разница между версиями — в использовании структуры *WSABUF* для получения данных. Вы можете предоставить один или несколько буферов *WSABUF*, указав их количество в *dwBufferCount* — в этом случае возможен комплексный ввод-вывод. Суммарное количество считанных байт передается в *lpNumberOfBytesRecvd*. При вызове функции *WSARecvFrom*, *lpFlags* может принимать следующие значения: 0 (при отсутствии параметров), *MSG\_DONTWAIT*, *MSG\_PEEK* или *MSG\_PARTIAL*. Данные флаги можно комбинировать логической операцией ИЛИ. Если при вызове функции задан флаг *MSG\_PARTIAL*, поставщик перешлет данные даже в случае приема лишь части сообщения. По возвращении флаг задается в *MSG\_PARTIAL*, только если сообщение принято частично. По возвращении *WSARecvFrom* присвоит параметру *lpFrom* (указатель на структуру *SOCKADDR*) адрес компьютера-отправителя. Опять же *lpFromLen* указывает на размер структуры *SOCKADDR*, однако в данной функции он является указателем на *DWORD*. Два последних параметра — *lpOverlapped* и *lpCompletionRoutine*, используются для перекрытого ввода-вывода (см. главу 8).

Другой способ приема (отправки) данных в сокетах, не требующих соединения, — установление соединения (хоть это и звучит странно). После создания сокета можно вызвать *connect* или *WSAConnect*, присвоив парамет-

пу *SOCKADDR* аДр^с удаленного компьютера, с которым необходимо связаться. Фактически никакого соединения не происходит. Адрес сокета, переданный в функцию соединения, ассоциируется с сокетом, чтобы было можно использовать функции *recv* и *WSARecv* вместо *recvfrom* или *WSARecvFrom* (поскольку источник данных известен). Если приложению нужно одновременно связываться лишь с одной конечной точкой, задействуйте возможность подключить сокет дейтаграмм.

## Отправитель

Есть два способа отправки данных через сокет, не требующий соединения. Первый и самый простой — создать сокет и вызвать функцию *sendto* или *WSASendTo*. Рассмотрим сначала функцию *sendto*:

```
int sendto(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
    int tolen
);
```

Параметры этой функции такие же, как и у *recvfrom*, за исключением *buf*— буфера данных для отправки, и *len* — показывающего сколько байт отправлять. Параметр *to* — указатель на структуру *SOCKADDR* с адресом принимающей рабочей станции.

Также можно использовать функцию *WSASendTo* из Winsock 2:

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Снова функция *WSASendTo* аналогична своей предшественнице. Она принимает указатель на одну или несколько структур *WSABUF* с данными для отправки получателю в виде параметра *lpBuffers*, а *dwBufferCount* задает количество структур. Для комплексного ввода-вывода можно отправить несколько структур *WSABUF*. Перед выходом *WSASendTo* присваивает четвертому параметру — *lpNumberOfBytesSent*, количество реально отправленных Получателю байт. Параметр *lpTo* — структура *SOCKADDR* для данного протокола с адресом приемника. Параметр *iToLen* — длина структуры *SOCKADDR*. Два последних параметра — *lpOverlapped* и *lpCompletionRoutine*, применяются для перекрытого ввода-вывода (см. также главу 8).



Как и при получении данных, сокет, не требующий соединения, можно подключать к адресу конечной точки и отправлять данные функциями *send* и *WSASend*. После создания этой привязки использовать для обмена данными функции *sendto* или *WSASendTo* с другим адресом нельзя — будет выдана ошибка *WSAEISCONN*. Отменить привязку сокета можно, лишь вызвав функцию *dosesocket* с описателем этого сокета, после чего следует создать новый сокет.

## Протоколы, ориентированные на передачу сообщений

Большинство протоколов, требующих соединения, — потоковые, а не требующих соединения — ориентированы на передачу сообщений. Поэтому при отправке и приеме данных нужно учесть ряд факторов. Во-первых, поскольку ориентированные на передачу сообщений протоколы сохраняют границы сообщений, данные, поставленные в очередь отправки, блокируются до завершения выполнения функции отправки. Если отправка не может быть завершена, при асинхронном или неблокирующем режиме ввода-вывода функция отправки вернет ошибку *WSAEWOULDBLOCK*. Это означает, что базовая система не смогла обработать данные и нужно попытаться вызвать функцию отправки повторно (подробней — в главе 8). Главное — в ориентированных на сообщения протоколах запись происходит только в результате самостоятельного действия.

С другой стороны, при вызове функции приема нужно предоставить вместительный буфер, иначе функция выдаст ошибку *WSAEMSGSIZE* — буфер заполнен, и оставшиеся данные отбрасываются. Исключением являются протоколы, поддерживающие обмен фрагментарными сообщениями, например AppleTalk PAP. Если была принята лишь часть сообщения, до возврата функции *WSARecvEx* присваивает параметру *flag* значение *MSGJPARTIAL*.

Для передачи дейтаграмм на основе протоколов, поддерживающих фрагментарные сообщения, задействуйте одну из функций *WSARecv*. При вызове *recv* нельзя отследить полноту чтения сообщения (эта задача программиста). После очередного вызова *recv* будет получена следующая часть дейтаграммы. Из-за данного ограничения удобнее использовать функцию *WSARecvEx*, позволяющую задавать и считывать флаг *MSGJPARTIAL*, который указывает на полноту чтения сообщения. Функции Winsock 2 *WSARecv* и *WSARecvFrom* также поддерживают работу с данным флагом.

В заключение рассмотрим, что происходит, когда сокет UDP явно привязан к локальному IP-интерфейсу. При использовании UDP-сокетов привязка к сетевому интерфейсу не выполняется. Вы создаете ассоциацию, посредством которой IP-интерфейс становится исходным IP-адресом отправляемых UDP-дейтаграмм. Физический интерфейс для передачи дейтаграмм фактически задает таблица маршрутизации. Если вместо *bind* вы вызываете *sendto* или *WSASendTo* или сначала устанавливаете соединение, сетевой стек автоматически выбирает наилучший локальный IP-адрес из таблицы маршрутизации. Таким образом, если сначала была выполнена привязка, исходный IP-адрес может быть неверным и не соответствовать интерфейсу, с которого фактически была отправлена дейтаграмма.

## Освобождение ресурсов сокета

Поскольку соединение не устанавливается, его формального разрыва или корректного закрытия не требуется. После прекращения отправки или получения данных отправителем или получателем просто вызывается функция *closesocket* с описателем требуемого сокета, в результате чего освобождаются все выделенные ему ресурсы.

## Пример

Теперь рассмотрим реальный код, выполняющий отправку и прием данных по протоколу. В листинге 7-3 приведен код приемника.

### Листинг 7-3. Приемник, не требующий установления соединения

```
// Имя модуля: Receiver.c
//
// Описание:
//   В данном примере выполняется получение UDP-дейтаграмм при помощи привязки к
//   конкретному интерфейсу и номеру порта, затем вызовы recvfrom0 блокируются.
//
// Параметры компиляции:
//   cl -o Receiver Receiver.c ws2_32.lib
//
// Параметры командной строки:
//   sender [-p:int] [-i:IP] [-n:x] [-b:x]
//   -p:int   Локальный порт
//   *i   -i:IP   Локальный IP-адрес, на котором будут прослушиваться соединения
//   -n:x     Количество попыток отправки сообщения
//   "" -b:     Размер буфера отправки
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT      5150
#define DEFAULT_COUNT     25
#define DEFAULT_BUFFER_LENGTH  4096

int  iPort      = DEFAULT_PORT;           // Порт, на котором будет идти прием
DWORD dwCount   = DEFAULT_COUNT;         // Количество читаемых сообщений
DWORD dwLength  = DEFAULT_BUFFER_LENGTH; // Длина приемного буфера
BOOL  bInterface = FALSE;                // Использование альтернативного интерфейса
char  szInterface[32];                    // Интерфейс, с которого читаются дейтаграммы

// Функция: usage
//
// Описание:
//   I   Выводит сведения о параметрах командной строки и выходит.
```

см. след. стр.

## Листинг 7-3.    (продолжение)

```

//
void usage()
{
    printf("usage: sender [-p:int] [-i:IP][-n:x] [-b:x]\n\n");
    printf("    -p:int    Local port\n");
    printf("    -i:IP     Local IP address to listen on\n");
    printf("    -n:x      Number of times to send message\n");
    printf("    -b:x      Size of buffer to send\n\n");
    ExitProcess(i);

// Функция: ValidateArgs
//
// Описание:
//     Анализирует параметры командной строки и задает
//     некоторые глобальные флаги для указания выполняемых действий
//
void ValidateArgs(int argc, char **argv)
{
    int            i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '.'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':    // Локальный порт
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'n':    // Количество попыток приема сообщения
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':    // Размер буфера
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'i':    // Интерфейс для приема дейтаграмм
                    if (strlen(argv[i]) > 3)
                    {
                        blInterface = TRUE;
                        strcpy(szInterface, &argv[i][3]);
                    }
                    break;
                'default':
                    usage();
            }
        }
    }
}

```

## Листинг 7-3. (продолжение)

```

        break;

//
Ц функция: main
//
// Описание:
//   Главный поток выполнения. Инициализирует Winsock, обрабатывает аргументы
//   командной строки, создает сокет, привязывает его к локальному интерфейсу и
//   порту, читает дейтаграммы.
int main(int argc, char **argv)

    WSADATA    wsd;
    SOCKET     s;
    char       *recvbuf = NULL;
    int        ret,
              il
    DWORD      dwSenderSize;
    SOCKADDR_IN sender,      * S/bt  'Mlit"\
                          local;

    // Анализ аргументов и загрузка Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2,2), &wsd) !=0)
    {
        printf("WSAStartup failed! \n");
        return 1;

    // Создание сокета и его привязка к локальному интерфейсу и порту
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; Xd\n", WSAGetLastError());
        return 1;
    }
    local.sin_family = AF_INET;
    local.sin_port = htons((short)iPort);
    if (bInterface)
        local.sin_addr.s_addr = inet_addr(szInterface);
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)

```

см. след. стр.

**Листинг 7-3.**    *(продолжение)*

```

    {
        printf("bind() failed: Xd\n", WSAGetLastError());
        return 1;
    }
    >
    // Выделение буфера приема
    //
    recvbuf = GlobalAlloc(GMEM_FIXED, dwLength);
    if (!recvbuf)
    {
У        printf("GlobalAlloc() failed: Xd\n", GetLastError());
        return 1;
    }
    // Чтение дейтаграмм
    //
    for(i = 0; i < dwCount; i++)
    <
        dwSenderSize = sizeof(sender);
        ret = recvfrom(s, recvbuf, dwLength, 0,
            (SOCKADDR *)&sender, &dwSenderSize);
        if (ret == SOCKET_ERROR)
        <
            printf("recvfrom() failed: Xd\n", WSAGetLastError());
            break;
    *   }
        else if (ret == 0)
            break;
        else
        {
            recvbuf[ret] = '\0';
            printf("[Xs] sent me: 'Xs'\n",
                inet_ntoa(sender.sin_addr), recvbuf);

        closesocket(s);

        GlobalFree(recvbuf);
        WSACleanup();
        return 0;
    }
    >

```

Прием дейтаграмм прост. Сначала необходимо создать сокет, затем привязать его к локальному интерфейсу. Для привязки к интерфейсу по умолчанию определите его IP-адрес функцией *getsockname*. В качестве параметра ей передается сокет, а она возвращает структуру *SOCKADDRIN*, которая указывает связанный с сокетом интерфейс. Затем для чтения входящих данных остается только выполнить вызовы *recvfrom*. Заметьте, что мы используем *recvfrom*, потому что нас не интересуют фрагментарные сообщения, так как протокол UDP не поддерживает их передачу. Фактически, стек TCP/IP пытается собрать большое сообщение из полученных фрагментов. Если один или



## Листинг 7-4. (продолжение)

```
//
// Функция: usage
// Описание:
// Выводит сведения о параметрах командной строки и выходит.

void usage()

    printf("usage: sender [-p:int] [-r:IP] "
           "[-c] [-n:x] [-b:x] [-d:c]\n\n");
    printfC      -p:int    Recipient's IP address or host name\n");
    printfC      -r:IP    Connect to remote IP first\n");
    printfC      -c        Number of times to send message\n");
    printfC      -n:x      Size of buffer to send\n");
    printfC      -b:x      Character to fill buffer with\n\n");
    printfC      -d:c      Character to fill buffer with\n\n");
    ExitProcess(i);

// Функция: ValidateArgs
//
// Описание:
// Анализирует параметры командной строки и задает
// некоторые глобальные флаги для указания выполняемых действий

void ValidateArgs(int argc, char **argv)

    ...

    for(i = 1; i < argc;
        // вад поаммЭ
        // bit),

        if <(argv[i][0] == '-' (argv[i][0]

            switch (tolower(argv[i][1]))

            <
            case 'p': // Удаленный порт
                if (strlen(argv[i]) > 3)
                    iPort = atoi(&argv[i][3]);
                break;
            case 'r': // IP-адрес получателя
                if (strlen(argv[i]) > 3)
                    strcpy(szRecipient, &argv[i][3]);
                break;
            case 'c': // Подключение к IP адресу получателя
                bConnect = TRUE;
                break;
            case 'n': // Количество попыток отправки сообщения
                if (strlen(argv[i]) > 3)
                    ^
                    $'
```

Листинг 7-4. (продолжение)

```

        dwCount = atol(&argv[i][3]);
        break;
    case 'b':          // Размер буфера
        if (strlen(argv[i]) > 3)
            dwLength = atol(&argv[i][3]);
        break;
    case 'd':          // Символ для заполнения буфера
        cChar = argv[i][3];
        break;
    default:
        usage();
        break;
}

// Функция: main
// Описание:
//   Главный поток выполнения. Инициализирует Winsock, обрабатывает аргументы
//   командной строки, создает сокет, при необходимости подключается по удаленному
//   IP-адресу, затем отправляет дейтаграммы получателю.
//
int main(int argc, char **argv)
{
    WSADATA wsd;
    SOCKET s;
    char *sendbuf = NULL;
    int ret;

    SOCKADDR_IN recipient;

    // Анализ аргументов и загрузка Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }

    // Создание сокета
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; Xd\n", WSAGetLastError());
    }
}

```

см. след. стр.



## Листинг 7-4. (продолжение')

```

        return 1;

// Разрешение IP-адреса или имени узла получателя

recipient.sin_family = AF_INET;
recipient.sin_port = htons((short)iPort);
if ((recipient.sin_addr.s_addr = inet_addr(szRecipient))
    == INADDR_NONE)

    struct hostent *host=NULL,

    host = gethostbyname(szRecipient),
    if (host)
        CopyMemory(&recipient.sm_addr, host->h_addr_list[0],
            host->h_length);
    else

,        printf("gethostbyname() failed: Xd\n", WSAGetLastErrorO);
        WSACleanupO;
        return 1;

// Выделение буфера отправки

sendbuf = GlobalAlloc(GMEM_FIXED, dwLength);
,. if (isendbuf)

(        printf("GlobalAlloc failed: Xd\n", GetLastErrorO);
        return 1;

memset(sendbuf, cChar, dwLength);

// Если задан параметр -с, выполняется "^дрючение" к отправителю
// и отправка данных функцией send(). $Щмет т , ь,
II
if (bConnect)
    if (connect(s, (SOCKADDR *)&recipient,
        sizeof(recipient)) == SOCKET.ERROR)

        printf("connect() failed Xd\n", WSAGetLastErrorO);
        GlobalFree(sendbuf),
        WSACleanupO,
        return 1;

for(i = 0; i < dwCount; i++)

        ret = send(s, sendbuf, dwLength, 0);  Щ
< Ч^\ ч ^м if (ret == SOCKET_ERROR)

```

**Листинг 7-4. (продолжение)**

```

        printf("send() failed: Xd\n", WSAGetLastError());
        break;
    }
    else if (ret == 0)
        break;
    // Функция send() отработала успешно!

else
{
    // Иначе используется функция sendtoQ
    //
    for(i = 0, i < dwCount; i++)
    {
        ret = sendto(s, sendbuf, dwLength, 0,
                     (SOCKADDR *)&recipient, sizeof(recipient));
        if (ret == SOCKET_ERROR)
        {
            printf("sendto() failed: Xd\n", WSAGetLastError());
            break;
        }
        else if (ret == 0)
            break;
        // Функция sendtoO отработала успешно!

    }

    closesocket(s);

    GlobalFree(sendbuf);
    WSACleanup();
    return 0;
}

```

## Дополнительные функции API

Рассмотрим API-функции Winsock, которые пригодятся **вам** при создании сетевых приложений

### Функция *getpeername*

Эта функция возвращает информацию об адресе сокета партнера на подключенном сокете

```

ir) t getpeername(
    SOCKET s,
    struct sockaddr FAR* name,
    mt FAR. namelen

```

Первый параметр — сокет для соединения, два последних — указатель на структуру *SOCKADDR* базового протокола и ее длина. Для сокетов дейтаграмм данная функция возвращает адрес, переданный вызову соединения (за исключением адресов, переданных в вызов *sendto* или *WSASendTo*).

## Функция *getsockname*

Эта функция противоположна *getpeername* и возвращает адресную информацию для локального интерфейса определенного сокета:

```
int getsockname(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
```

Используются те же параметры, что и для *getpeername*, однако возвращается информация о локальном адресе. В случае TCP адрес совпадает с сокетом сервера, слушающим на заданном порте и IP-интерфейсе.

## Функция *WSADuplicateSocket*

Данная функция применяется для создания структуры *WSAPROTOCOLINFO*, которую можно передать другому процессу, что позволит ему открыть описатель того же базового сокета и оперировать данным ресурсом. Заметьте: такая необходимость возникает только между процессами. Поток в одном и том же процессе могут свободно передавать описатели сокета. Функция определена так:

```
int WSADuplicateSocket(
    SOCKET s,
    DWORD dwProcessId,
    LPWSAPROTOCOL_INFO lpProtocolInfo
);
```

Первый параметр — копируемый описатель сокета. Второй — *dwProcessId*, код процесса, который будет использовать скопированный сокет. Третий параметр — *lpProtocolInfo*, указатель на структуру *WSAPROTOCOLINFO*, которая содержит информацию, необходимую целевому процессу для открытия копии описателя. Некоторые виды межпроцессного взаимодействия должны происходить таким образом, чтобы текущий процесс мог передать структуру *WSAPROTOCOLINFO* целевому, который затем использует ее для создания описателя сокета (при помощи функции *WSASocket*).

Описатели в обоих сокетах можно использовать для ввода-вывода независимо, однако Winsock не обеспечивает контроля за доступом, поэтому программисту необходимо предусмотреть некие способы синхронизации. Все сведения о состоянии любого сокета хранятся в одном месте для всех его описателей, так как копируются описатели сокета, а не сам сокет. Например, любой параметр сокета, заданный функцией *setsockopt* для одного из описателей, затем можно увидеть, вызвав функцию *getsockopt* для любого другого его описателя. Если процесс вызывает *closesocket* для копии сокета, описатель

в данном процессе освобождается, однако сокет останется открытым, пока функция *closesocket* не будет вызвана для последнего его описателя.

Кроме того, учтите общие особенности сокетов при уведомлении с использованием *WSAAsyncSelect* и *WSAEventSelect*. Эти функции применяются для асинхронного ввода-вывода (см. главу 8). При их вызове с любым из общих описателей отменяется регистрация любого предыдущего события для сокета, независимо от того, какой описатель применялся для регистрации. Поэтому, например, через общий сокет нельзя доставить события *FDREAD* процессу А и события *FDJWRITE* процессу В. Если необходимо передавать уведомления о событиях по обоим описателям, измените конструкцию приложения, используя потоки вместо процессов.

## Функция *TransmitFile*

Это расширение Microsoft позволяет быстро передавать данные из файла. Высокая эффективность обусловлена тем, что вся передача данных происходит в режиме ядра. Если приложение считывает блок данных из файла, а затем вызывает *send* или *WSASend*, происходят многократные переключения между режимами ядра и пользовательским. При вызове *TransmitFile* весь процесс чтения и отправки выполняется в режиме ядра. Функция определена таю

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    DWORD dwFlags
);
```

Параметр *hSocket* — подключенный сокет, по которому будет передан файл. Параметр *hFile* — описатель открытого файла. Параметр *nNumberOfBytesToWrite* задает количество записываемых из файла байт. Если он равен 0, файл отправляется целиком. Параметр *nNumberOfBytesPerSend* задает размер отправляемых блоков данных для операций записи. Например, если он равен 2048, *TransmitFile* передаст файл порциями по 2 кб; если 0 — используется стандартный размер отправки. Параметр *lpOverlapped* определяет структуру *OVERLAPPED*, применяемую в перекрытом вводе-выводе (см. также главу 8).

Следующий параметр — *lpTransmitBuffers*, представляет собой структуру *TRANSMIT\_FILE\_BUFFERS*, содержащую данные, которые нужно отправить до и после передачи файла:

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS-
```

Поле *Head* — указатель на данные, которые отправляются перед передачей файла. Поле *HeadLength* задает количество заранее передаваемых данных. Поле *Tail* ссылается на данные, отправляемые после передачи файла. В *TailLength* указано количество передаваемых затем байт.

Последний параметр — *dwFlags*, управляет режимами работы *TransmitFile*. Вот описание используемых в нем флагов.

**III *TF\_DISCONNECT*** — инициирует закрытие сокета после передачи данных.

**III *TF\_REUSE\_SOCKET*** — позволяет повторно использовать в функции *AcceptEx* описатель сокета в качестве клиентского сокета.

**K *TF\_USE\_DEFAULT\_WORKER* и *TF\_USE\_SYSTEM\_THREAD*** — указывают, что передача должна идти в контексте стандартного системного процесса. Эти флаги полезны при передаче больших файлов.

- ***TF\_USE\_KERNEL\_APC*** — указывает, что передача должна выполняться ядром при помощи *асинхронных вызовов процедур* (Asynchronous Procedure Call, APC). Это существенно увеличивает производительность, если для считывания файла в кэш требуется лишь одна операция чтения.
- ***TF\_WRITEBEHIND*** — указывает, что *TransmitFile* может завершиться, не получив подтверждений о приеме данных от удаленной системы.

## Для платформы Windows CE

Вся информация из предыдущих разделов в равной степени относится к Windows CE. Исключение — функции, специфичные для Winsock 2, поскольку Windows CE опирается на спецификацию Winsock 1.1 — например WSA-разновидностей функций. В Windows CE доступны только следующие WSA-функции: *WSAStartup*, *WSACleanup*, *WSAGetLastError* и *WSAIocctl*. Мы уже обсуждали первые три из них, а о последней поговорим в главе 9.

Windows CE поддерживает протокол TCP/IP, следовательно, у вас есть доступ как к TCP, так и к UDP. Помимо TCP/IP поддерживаются инфракрасные сокеты. Протокол IrDA поддерживает только потоковые соединения. При использовании обоих протоколов выполняют все обычные API-вызовы Winsock 1.1 для подготовки сокетов и передачи данных. Необходимо учитывать ошибку в Windows CE 2.0, связанную с дейтаграммными UDP-сокетами: вызов функций *send* или *sendto* влечет утечки памяти ядра. Эта ошибка исправлена в Windows CE 2.1, но из-за того, что ядро записано в ПЗУ, в Windows CE 2.0 невозможно устранить данную проблему при помощи распространяемых программных обновлений. Единственное решение — отказаться от использования дейтаграмм в Windows CE 2.0.

Windows CE не поддерживает консольные приложения и использует только кодировку UNICODE, поэтому примеры, представленные в данной главе, предназначены для Windows 95, 98, NT и 2000. Мы приводим их, чтобы дать вам возможность изучить основные концепции Winsock без утомительного рассмотрения программного кода. Почти всегда необходим пользовательский интерфейс, если только вы не пишете службу для Windows CE — тогда потребуется создать множество дополнительных функций для обработчиков

событий окон и других элементов пользовательского интерфейса, разбор которых помогает вам понять главные аспекты применения Winsock

Также существует дилемма использовать функции Winsock, поддерживающие UNICODE, или нет. Выбор кодировки лежит на программисте Winsock все равно, что вы передаете функциям: лишь бы это был действительный буфер (конечно, нужно привести буфер к соответствующему типу, чтобы не появлялись предупреждения при компиляции). Если вы приведете строку UNICODE к *char* ', не забудьте соответственно изменить параметр длины, задающий количество отправляемых байт. Для правильного отображения любых отправленных или принятых данных в Windows CE необходимо убедиться, что они в кодировке UNICODE. Это нужно и для любых других функций Win32, требующих строк в кодировке UNICODE. В общем, создание приложений Winsock в Windows CE более трудоемко.

Для компиляции и запуска приведенных примеров в Windows CE требуется незначительно изменить код. Заголовочным файлом будет Winsock.h, в отличие от Winsock2.h. Функция *WSAStartup* должна загружать версию Winsock 1.1, потому что она текущая в Windows CE. Эта ОС не поддерживает консольных приложений, поэтому необходимо использовать функцию *WinMain* вместо *main*. Не требуется включать окно в приложение, просто не используйте функции консольного текстового ввода-вывода, например *printf*.

## Другие семейства адресов

Все API-функции Winsock, представленные в этой главе, не зависят от протокола. Поэтому их легко можно применять и для других протоколов, поддерживаемых платформами Win32. Следующие разделы объясняют примеры клиент-серверного кода для других семейств протоколов, которые находятся на прилагаемом компакт-диске.

### Протокол AppleTalk

Единственный пример, связанный с AppleTalk, представлен для иллюстрации основных клиент-серверных технологий и поддерживает как PAP, так и ADSP. Протокол PAP ориентирован на сообщения, не требует соединения, и не надежен. Этим он похож на UDP, но есть и два важных отличия. PAP поддерживает фрагментарные сообщения, то есть функция *WSARecvEx* может вернуть лишь часть дейтаграммного сообщения. При этом необходимо проверить значение флага *MSG\_PARTIAL*, чтобы узнать, нужно ли дополнительно вызывать функцию для получения остатков сообщения. Кроме того, необходимо задавать специфичные для протокола PAP параметры сокета перед каждым чтением (Подробнее о параметре *SO\_J4UML\_READ*, используемом совместно с функцией *setsockopt*, — в главе 9.) Ознакомьтесь с примером *Atalk.c* на прилагаемом компакт-диске, где иллюстрируется проверка флага *MSG\_PARTIAL* и порядок использования параметра *SOPRIMER\_READ*.

Протокол ADSP требует соединения, потоковый, надежный и во многом похож на TCP. API-вызовы для AppleTalk практически не отличаются от представленных в этой главе примеров для UDP и TCP. Разница лишь в разре-

нии имен. Помните, что для AppleTalk перед поиском или регистрацией имени необходимо выполнить привязку к пустому адресу (см. главу 6).

У протокола AppleTalk есть ограничение: его поддержка появилась в Winsock 1.1, а когда был разработан Winsock 2, оказалось, что с AppleTalk не работают некоторые новые функции. Использование любой из *WSASend*- или *WSARecv*-функций может привести к непредсказуемым результатам, например к возврату отрицательного количества байт. Данная проблема изложена в статье Q164565 базы знаний Microsoft Knowledge. Исключение составляет функция *WSARecvEx*, которая просто вызывает *recv*, а в параметре ввода-вывода/я<sup>^</sup> можно просмотреть значение флага *MSG\_PARTIAL* после выхода.

## Инфракрасные сокеты

Поддержка IrDA появилась недавно, в Windows CE, 98 и 2000. Протокол IrDA требует соединения, потоковый и надежный. Снова вопрос в разрешении имен, которое значительно отличается от принятого в IP. Необходимо знать и еще одно отличие: на платформах Windows CE можно использовать только функции Winsock 1.1 для инфракрасных сокетов, так как Windows CE поддерживает только спецификацию Winsock 1.1. В Windows 98 и 2000 допустимо применение функций, специфичных для Winsock 2. Код примера использует только функции Winsock 1.1. В Windows 98 и 2000 необходимо загрузить библиотеку Winsock 2.2 или более позднюю, потому что поддержка семейства адресов *AF\_IRDA* в более ранних версиях не доступна.

Пример кода для инфракрасных сокетов приведен в файлах *Ircommon.h*, *Ircommon.c*, *Irclient.c* и *Irserver.c*. Первые два файла просто определяют две общие функции: одну — для отправки, другую — для приема данных. Они используются как клиентским, так и серверным приложениями. Клиентская часть подробно расписана в файле *Irclient.c*. Сначала нумеруются все устройства в зоне видимости. Затем следуют попытки подключиться к каждому из них с заданным именем службы. Дальнейшая работа ведется с первым устройством, принявшим соединение. Клиент отправляет данные и считывает их обратно. Серверная часть описана в файле *Irserver.c*. Сервер создает инфракрасный сокет, выполняет привязку заданного имени службы к сокету и ожидает подключений клиентов. Для каждого клиента порождается поток для приема данных и отправки их обратно.

Заметьте: данные примеры написаны для Windows 98 и 2000. Подобно примерам по TCP/IP, их нужно немного изменить для запуска в Windows CE. Также следует учесть два уже упоминавшихся ограничения Windows CE: эта ОС не поддерживает консольные приложения и требует кодировать строки в UNICODE.

## Интерфейс с NetBIOS

В главе 1 мы говорили, что NetBIOS совместима с несколькими разными транспортом, применяемыми в Winsock, а в главе 6 — обсудили, как регистрировать совместимые с NetBIOS транспорты и сокеты на основе любого из них. Для каждой комбинации «протокол — адаптер» есть две записи: *SOCK\_DGRAM* и *SOCK\_SEQPACKET*, соответствующих не требующим соединения дейтаграммам

и потоковым сокетами, которые весьма похожи на сокеты UDP и TCP. Кроме решения имен работа с Wmsock-интерфейсом NetBIOS ничем не отличается от описанной в этой главе. Помните, что хорошо написанный сервер должен прослушивать все доступные LANA, а клиент со своей стороны — пытаться установить соединение по всем LANA.

Первые два примера на компакт-диске — *Wsnbsvr.c* и *Wsnbclnt.c*. Они используют потоковые сокеты *SOCK\_SEQPACKET*. Сервер создает сокет для всех LANA, которые нумеруются функцией *WSAEnumProtocols*, и привязывает его к стандартному имени сервера. После установления клиентом соединения, сервер создает поток для его обслуживания. С этого момента поток просто читает входящие данные и возвращает их клиенту. Аналогично, клиент пытается подключиться по всем LANA. После первого успешного соединения другие сокеты закрываются. Затем клиент отправляет данные серверу, а сервер снова их возвращает.

В файле *Wsnbdgs.c* приведен код для отправки и приема дейтаграммных сообщений (через сокет с типом *SOCK\_DGRAM*). Этот протокол не требует соединения, поэтому для отправки сообщений серверу используются все доступные транспорты, так как заранее неизвестно, какие из них поддерживает сервер. Кроме того, в примере реализована поддержка уникальных, групповых и широковещательных сообщений (см. главу 1).

## Протокол IPX/SPX

В файле *Sockspx.c* приведен пример использования протокола IPX, а также потокового и последовательного SPXII. В одном примере реализован отправитель и приемник для всех трех протоколов. Конкретный протокол задается при помощи параметра *-p* в командной строке. Пример прост и легок в освоении. Функция *main* анализирует аргументы, а затем вызывает функцию *Server* или *Client*. Для протокола SPXII, ориентированного на соединения, это значит, что сервер привязывает сокет к адресу внутренней сети и ожидает соединений с клиентом, который, в свою очередь, пытается установить соединение с указанным в командной строке сервером. После установления соединения отправка и прием данных идут обычным образом.

Для не требующего соединения протокола IPX пример еще проще. Сервер просто выполняет привязку к внутренней сети и ожидает входящие данные, вызывая функцию *recvfrom*. Клиент отправляет данные получателю, указанному в командной строке, функцией *sendto*.

Два раздела этого примера требуют разъяснения. Функция *FillIpxAddress* отвечает за кодирование указанного в командной строке в кодировке ASCII IPX-адреса в структуру *SOCKADDR\_IPX*. Как упоминалось в главе 6, IPX-адреса представляются в виде шестнадцатеричных строк, то есть каждый символ адреса фактически занимает 4 бита в адресных полях структуры *SOCKADDR\_IPX*. Функция *FillIpxAddress* получает IPX-адрес и вызывает функцию *AtoH*, которая и выполняет преобразование.

Еще одна функция — *EnumerateAdapters*, выполняется, если в командной строке задан флаг *-m*. Для подсчета количества доступных локальных IPX-адресов она использует параметр сокета *IPX\_MAX\_ADAPTER\_NUM*, а затем для



получения каждого адреса вызывает параметр сокета *IPXADDRESS*. Мы используем их, потому что в нашем примере IPX-адреса заданы напрямую, раз-решение имен не выполняется (подробнее — в главах 9 и 10).

## Протокол АТМ

Протокол АТМ доступен в Winsock под управление Windows 98 и 2000. Пример для АТМ состоит из файлов *Wsocketm.c*, *Support.c*, and *Support.h*. Два последних файла содержат вспомогательные подпрограммы, используемые в *Wsocketm.c* и обеспечивающие регистрацию локальных АТМ-адресов и их перекодирование. АТМ-адреса шестнадцатеричные, как и IPX-адреса, поэтому мы применяем уже знакомую функцию *AtoH*. Для получения количества локальных АТМ-интерфейсов используется команда *SIO\_GET\_NUMBERJDF\_ATM\_DEVICES*, а затем при помощи команды *SIO\_GET\_ATM\_ADDRESS* мы получаем фактический адрес (подробнее — в главе 9)–

В отличие от описанных ранее примеров, клиентское и серверное при-ложение представлено одним файлом — *Wsocketm.c*. Так как АТМ поддержи-вает только подключения, требующие соединения, пример небольшой и большинство кода находится в функции *main*. Сервер выполняет привязку к определенному локальному интерфейсу и ожидает подключений клиентов, которые обрабатываются в том же потоке, что и слушающий сокет. Это оз-начает, что сервер может одновременно обслуживать только одного клиен-та. Клиент вызывает *connect* с АТМ-адресом сервера. После установления со-единения начинается обмен данными.

А теперь несколько предостережений. После вызова сервером функции *WSAAccept* будет выведен адрес клиента. Однако к моменту получения сервером запроса на соединение адрес клиента еще неизвестен, потому что функ-ция *accept* не устанавливает соединение до конца. Это справедливо и для кли-ентской стороны — клиентский вызов для установления соединения с сервером признается успешным, даже если соединение до конца не установлено. Это означает, что после вызова *connect* или *accept* немедленная отправка мо-жет сорваться без выдачи предупреждений. К сожалению, приложение не спо-собно узнать, когда соединение полностью готово к использованию. Вдоба-вок, АТМ поддерживает только резкое завершение: при вызове функции *closesocket* соединение немедленно разрывается. Для протоколов, не поддержи-вающих плавное завершение, при вызове *closesocket* данные, ожидающие в очереди на любом конце соединения, обычно отбрасываются. Это вполне приемлемое поведение. Впрочем, поставщик АТМ действует предусмотрительно: при закрытии одной из сторон своего сокета во время передачи данных, Winsock все же возвращает данные из приемной очереди этого сокета.

## Резюме

В этой главе обсуждались основные функции Winsock, необходимые для обмена данными по различным протоколам. Мы предоставили данную ин-формацию в контексте только одной модели ввода-вывода: блокирующих сокетов. В следующей главе мы рассмотрим другие модели ввода-вывода, доступные в Winsock.

## Г Л А В А

# Ввод-вывод в Winsock

Эта глава посвящена управлению вводом-выводом через сокеты в Windows-приложениях. В Winsock такое управление реализовано с помощью режимов работы и моделей ввода-вывода. *Режим* (mode) сокета определяет поведение функций, работающих с сокетом. *Модель* (model) сокета описывает, как приложение производит ввод-вывод при работе с сокетом. Модели не зависят от режима работы и позволяют обходить их ограничения.

Winsock поддерживает два режима: *блокирующий* (blocking) и *неблокирующий* (nonblocking). Эти режимы подробно описаны в начале главы, здесь же демонстрируется их использование в приложениях для управления вводом-выводом. Далее приведен ряд интересных моделей, которые помогают приложению управлять несколькими сокетами одновременно в асинхронном режиме: *select*, *WSAAsyncSelect*, *WSAEventSelect*, *перекрытый ввод-вывод* (overlapped I/O) и *порт завершения* (completion port). В конце главы рассматриваются достоинства и недостатки различных режимов и моделей, а также обсуждается выбор вариантов, наиболее подходящих для конкретных случаев.

Все платформы Windows поддерживают блокирующий и неблокирующий режимы работы сокета. Впрочем, конкретная платформа может поддерживать не все модели. В Windows CE доступна лишь одна модель ввода-вывода, в Windows 98 и Windows 95 — большинство моделей, кроме портов завершения (поддержка конкретной модели зависит от версии Winsock). В Windows NT и Windows 2000 доступны все модели (табл. 8-1).

**Табл. 8-1. Поддерживаемые модели ввода-вывода**

Платформа	<i>Select Async</i>	<i>WSA-Event Select</i>	<i>WSA-ввод-Select</i>	Перекрытый Port вывод	Порт завершения
WindowsCE	Да	Нет	Нет	Нет	Нет
Windows 95 (Winsock 1)	Да	Да	Нет	Нет	Нет
Windows 95 (Winsock 2)	Да	Да	Да	Да	Нет
Windows 98	Да	Да	Да	Да	Нет
Windows NT	Да	Да	Да	Да	Да
Windows 2000	Да	Да	Да	Да	Да

## Режимы работы сокетов

В блокирующем режиме функции ввода-вывода, такие как *send* и *recv*, перед завершением ожидают окончания операции. В неблокирующем — работа функций завершается немедленно. Приложения, выполняемые на платформах Windows CE и Windows 95 (в случае Winsock 1), поддерживают очень мало моделей ввода-вывода и требуют от программиста описать блокирование и разблокирование сокетов в разных ситуациях.

### Блокирующий режим

При блокировке сокета необходима осторожность, так как этом режиме любой вызов функции Winsock именно блокирует сокет на некоторое время. Большинство приложений Winsock следуют модели «поставщик — потребитель», в которой программа считывает или записывает определенное количество байт и затем выполняет с ними какие-либо операции (листинг 8-1).

Листинг 8-1. Простейший пример блокировки сокета

```
SOCKET sock;  
char buff[256];  
int done = 0;  
  
while(!done)  
{  
    nBytes = recv(sock, buff, 65);  
    if (nBytes == SOCKET_ERROR)  
    {  
        printf("recv failed with error %d\n",  
              WSAGetLastErrorQ);  
        Return;  
    }  
    OoComputationOnData(buff);  
}
```

Проблема в том, что функция *recv* может не завершиться никогда, так как для этого нужно считать какие-либо данные из буфера системы. В такой ситуации некоторые программисты могут соблазниться «подглядыванием» данных (чтение без удаления из буфера), используя флаг *MSG\_PEEK* в *recv* или вызывая *ioctlsocket* с параметром *FIONREAD*. Подобный стиль программирования заслуживает резкой критики. Издержки, связанные с «подглядыванием», велики, так как необходимо сделать один или более системных вызовов для определения числа доступных байт, после чего все равно придется вызывать *recv* для удаления данных из буфера.

Чтобы этого избежать, следует предотвратить замораживание приложения из-за недостатка данных (из-за сетевых проблем или проблем клиента)

без постоянного «подглядывания» в системные сетевые буферы. Один из методов — разделить приложения на считывающий и вычисляющий потоки, совместно использующие общий буфер данных. Доступ к буферу регулируется синхронизирующим объектом, таким как событие или мьютекс. Задача считывающего потока — постоянно читать данные из сети и помещать их в общий буфер. Считав минимально необходимое количество данных, этот поток инициирует сигнальное событие, уведомляющее вычисляющий поток, что можно начинать вычисления. Затем вычисляющий поток удаляет часть данных из буфера и производит с ними необходимые операции. В листинге 8-2 реализованы две функции: для приема данных (*ReadThread*) и их обработки (*ProcessThread*).

### Листинг 8-2. Пример многопоточного программирования в режиме блокировки

```
// Перед созданием двух потоков,
// инициализируется общий буфер (data)
// и создается сигнальное событие (hEvent)
CRITICAL_SECTION data;
HANDLE          hEvent;
TCHAR          buff[MAX_BUFFER_SIZE];
int             nbytes;

// Считывающий поток
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
                nLeft);
            if (nRead == -1)
            {
                perror("error\n");
                ExitThread0;

                nTotal += nRead;
                nLeft -= nRead;

                nBytes += nRead;
            }
        }
    }
}
```

**Листинг 8-2.** *(продолжение)*

```

        LeaveCriticalSection(&data),
    >
    SetEvent(hEvent);

// Вычисляющий поток
void ProcessThread(void)
{
    WaitForSingleObject(hEvent),

    EnterCriticalSection(&data),
    DoSomeComputationOnData(buff),

    // Удаление обработанных данных из буфера
    // и сдвиг оставшихся в начало массива
    nBytes -= NUM_BYTES_REQUIRED,

    LeaveCriticalSection(&data),
}

```

Недостаток блокировки сокетов — приложению трудно поддерживать связь по нескольким сокетам одновременно. Приведенную схему можно изменить, чтобы считывающий и вычисляющий потоки создавались отдельно для каждого сокета. Для этого придется попотеть, зато вы получите более эффективное решение. Но учтите: данный вариант не предусматривает масштабирования, если вам придется работать с большим количеством сокетов.

## Неблокирующий режим

Альтернатива описанному режиму — режим без блокировки. Он несколько сложнее в использовании, но обеспечивает те же возможности, что и режим блокировки, плюс некоторые преимущества. В листинге 8-3 показано, как создать сокет и перевести его в неблокирующий режим.

**Листинг 8-3. Создание сокета без блокировки**

```

SOCKET      s,
unsigned long ul = 1,
int         nRet,

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIONBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    // если не удалось перевести сокет в неблокирующий режим
}

```

Если сокет находится в неблокирующем режиме, функции Winsock завершаются немедленно. В большинстве случаев они будут возвращать ошибку.

*WSAEWOULDBLOCK*, означающую, что требуемая операция не успела завершиться за время вызова. Например, вызов *recv* вернет *WSAEWOULDBLOCK*, если в системном буфере нет данных. Часто функцию требуется вызывать несколько раз подряд, пока не будет возвращен код успешного завершения. Вот список возвращаемых значений *WSAEWOULDBLOCK* при вызове наиболее часто встречающихся функций Winsock:

- *WSAAccept* и *accept* — приложение не получило запрос на соединение, повторите вызов для проверки наличия запросов,
- *closesocket* — скорее всего, функция *setsockopt* была вызвана с параметром *SOJINGER* и задан ненулевой тайм-аут,
- *WSAConnect* и *connect* — соединение инициировано, для проверки завершения повторите вызов,
- *WSARecv*, *recv*, *WSARecvFrom* и *recvfrom* — данные не были приняты, повторите проверку позже,

**III *WSASend*, *send*, *WSASendTo* и *sendto*** — в буфере нет места для записи исходящих данных, попробуйте вызвать функцию позже

Так как большинство вызовов в неблокирующем режиме будут возвращать ошибку *WSAEWOULDBLOCK*, анализируйте все возвращаемые коды ошибок и будьте готовы прервать операцию в любой момент. К сожалению, многие программисты непрерывно вызывают функцию до успешного завершения. Между тем, для чтения 200 байт данных создание цикла, состоящего лишь из вызова *recv*, ничуть не лучше, чем вызов в блокирующем режиме с флагом *MSG\_PEEK*, обсуждавшийся ранее. Модели ввода-вывода Winsock могут помочь приложению определить, когда сокет доступен для чтения и записи.

У каждого режима работы сокета свои достоинства и недостатки. Режим блокировки проще концептуально, но в нем сложнее обрабатывать несколько сокетов одновременно или нерегулярные потоки данных. С другой стороны, режим без блокировки требует больше кода для обработки ошибки *WSAEWOULDBLOCK* в каждом вызове. Модели ввода-вывода сокетов помогают приложению асинхронно обрабатывать соединения на одном и более сокетах.

## Модели ввода-вывода сокетов

Приложения Winsock могут использовать пять моделей для управления вводом-выводом: *select*, *WSAAsyncSelect*, *WSAEventSelect*, перекрытый ввод-вывод и порты завершения. В этом разделе объясняются особенности каждой модели и основные принципы управления сокетами. На прилагаемом компакт-диске приведены примеры приложений, показывающие разработку простейшего TCP-сервера с учетом принципиальных особенностей каждой модели.

### Модель *select*

а модель наиболее широко доступна в Winsock. Мы называем ее *select*, по-

У что управление вводом-выводом в ней основано на использовании Ункц *select*. Идея восходит к модели сокетов Беркли для Unix. Эта модель

была встроена в Winsock 1.1, чтобы дать приложениям, избегающим блокировки при вызове сокета, возможность управлять несколькими сокетами в определенном порядке. Так как Winsock версии 1.1 совместим с Беркли приложение, использующее сокеты Беркли и функцию *select*, в принципе может выполняться в Windows без модификации.

Функцию *select* используют и для определения, есть ли в соquete данные и можно ли туда записать новые. Основная цель функции — избежать блокировки приложения при связанных с вводом-выводом вызовах, например *send* или *recv*, когда сокет работает в блокирующем режиме, и предотвратить появление ошибки *WSAEWOULDBLOCK* — в неблокирующем. Функция *select* блокирует операции ввода-вывода, пока не будут соблюдены условия, заданные в качестве параметров. Прототип функции *select*:

```
int select(  
    int nfdс,  
    fd_set FAR * readfdс,  
H   fd_set FAR * writefdс,  
    fd_set FAR * exceptfdс,  
    const struct timeval FAR * timeout  
);
```

Первый параметр — *nfdс*, игнорируется, он включен лишь для совместимости с приложениями Беркли. Заметьте, что есть три параметра с типом *fdset*: один для проверки возможности чтения — *readfdс*, другой для проверки возможности записи — *writefdс* и третий для *срочных* (out of band, OOB) данных — *exceptfdс*. Тип *fdset* представляет набор сокетов. Набор *readfdс* определяет сокеты, удовлетворяющие одному из следующих условий:

- данные доступны для чтения;

III соединение закрыто, сброшено или завершено;

- если вызвать функцию *listen*, когда соединение находится в состоянии ожидания, вызов функции *accept* будет успешным.

Набор *writefdс* определяет сокеты, удовлетворяющие одному из следующих условий-

III возможна отправка данных;

- если обрабатывается неблокирующий вызов соединения, попытка соединения удалась.

Наконец, набор *exceptfdс* определяет сокеты, удовлетворяющие одному из следующих условий:

- если обрабатывается неблокирующий вызов соединения, попытка соединения не удалась;

III ООВ-данные доступны для чтения.

Например, чтобы проверить возможность чтения из сокета, добавьте его в набор *readfdс* и подождите завершения функции *select*. Затем проверьте, входит ли еще этот сокет в набор *readfdс*. Если да, то сокет доступен для чтения и можно работать с его данными. Любые два из трех параметров

*fSy exceptfds*) могут быть *NULL* (но хотя бы один должен быть не *NULL*). Любой ненулевой набор должен содержать хотя бы один описатель сокета, иначе функции *select* будет нечего ожидать. Последний параметр — *timeout*, представляет собой указатель на структуру *timeval*, определяющую, сколько времени *select* будет ждать окончания ввода-вывода. Если *timeout* равен *NULL*, *select* будет ждать, пока не найдет хотя бы один описатель, отвечающий заданному критерию. Структура *timeval* определена так:

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

Поле *tv\_sec* задает время ожидания в секундах, а поле *tv\_usec* — в миллисекундах. Тайм-аут {0, 0} означает, что функция *select* должна завершаться немедленно, позволяя приложению определить ее результат. Впрочем, этого следует избегать. При успешном завершении *select* возвращает в структуру *paxfd\_set* общее количество описателей сокетов, у которых есть ожидающие операции ввода-вывода. Если время *timeval* истекает, возвращается 0. В случае любой неудачи *select* возвращает *SOCKETJIROR*.

Перед тем как отслеживать сокеты с помощью *select*, ваше приложение должно сформировать одну или все *структуры fd\_set*, присвоив набору описатели. Добавив сокет в один из наборов, вы сможете узнать, происходила ли конкретная операция ввода-вывода с этим сокетом. В Winsock определены следующие макросы для работы с наборами *#\_set*

- ***FD\_CLR(s, \*set)*** — удаляет сокет *s* из набора *set*;

- III ***FD\_ISSET(s, \*set)*** — проверяет, входит ли сокет *s* в набор *set*;

- II ***FD\_SET(s, \*set)*** — добавляет сокет *s* в набор *set*;

- III ***FDZERO(\*set)*** — инициализирует *set* как пустой набор.

Например, если вы хотите узнать, можно ли читать данные из сокета без блокировки, добавьте его в набор *fdjread* при помощи макроса *FD\_SET* и вызовите *select*. Чтобы проверить, остался ли сокет в этом наборе, используйте макрос *FDJSSET*. Вот типичный алгоритм применения *select* для работы с одним или несколькими сокетами.

- 1 • Инициализируйте все интересующие *vacfdset* макросом *FDZERO*.
- Добавьте описатели сокетов в соответствующие наборы *fdjset* макросом

Вызовите функцию *select* и дождитесь результата: *select* вернет общее количество описателей, оставшихся в наборах, и соответственно обновит сами наборы.

используя результат работы *select*, приложение может определить, какие сокеты осуществляют ввод-вывод в данное время, проверяя каждый *fd\_set* макросом *FD\_ISSET*.



- 5 Выявив активные сокеты, обработайте их ввод-вывод и продолжите с шага 1

По завершении работы функция *select* удаляет из каждой структуры/*fd\_set* описатели сокетов, не участвующих в операциях ввода-вывода. Этим объясняется необходимость использовать макросы *FDISSET* шаге 4, чтобы определить, является ли конкретный сокет частью набора. В листинге 8-4 показаны основные этапы реализации модели *select* для одного сокета. Для нескольких сокетов нужно обработать список или массив дополнительных сокетов.

#### Листинг 8-4. Применение модели *select* для управления вводом-выводом через сокет

```

SOCKET s,
fd_set fdread,
int ret,

// Создание сокета и установление соединения

// Управление вводом-выводом сокета
while(TRUE)
{
    // Всегда очищайте набор перед вызовом
    // select0
    FD_ZERO(&fdread),

    // Добавление сокета s к набору для проверки чтения

    if ((ret = select(0, &fdread, NULL, NULL, NULL))
        == SOCKET_ERROR)

        //I Обработка ошибки

    if (ret > 0)
    {
        // В этом простейшем случае select0 должна вернуть 1
        //I Приложение, работающее с несколькими сокетами,
        // может получить большую величину
        // Здесь должна быть проверка,
        // входит ли сокет в набор

        if (FD_ISSET(s, &fdread))
        {
            // Через сокет s идет чтение
        }
    }
}

```

## Модель *WSAAsyncSelect*

Winsock поддерживает полезную асинхронную модель ввода-вывода, позволяющую приложению получать информацию о событиях, связанных с сокетом, при помощи сообщений Windows. Это достигается вызовом функции *WSAAsyncSelect* после создания сокета. Данная модель первоначально появилась в приложениях Winsock 1.1 для облегчения взаимодействия приложений в многозадачной среде 16-битных платформ, таких как Windows for Workgroups. Но она полезна и для современных приложений, особенно если они обрабатывают сообщения окон в стандартной процедуре *{ivmproc}*. Эта модель также используется объектом *CSocket* из библиотеки классов Microsoft (Microsoft Foundation Class, MFC).

### Уведомления о сообщениях

Прежде чем использовать модель *WSAAsyncSelect*, приложение должно создать окно, используя функцию *CreateWmdow*, и процедуру обработки сообщений (*ivmproc*) для этого окна. Можно использовать диалоговое окно с диалоговой процедурой (так как это частный случай окна). Здесь достаточно продемонстрировать простое окно с дополнительной процедурой. Создав инфраструктуру окна, вы вправе создавать сокеты и активизировать уведомления вызовом функции *WSAAsyncSelect*.

```
int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
```

Параметр *s* — интересующий нас сокет. Параметр *hWnd* — описатель окна (или диалога), которое должно получить уведомление, когда произойдет сетевое событие. Параметр *wMsg* определяет сообщение, которое будет в этом случае отправлено окну с описателем *hWnd*. Обычно сообщению присваивается код выше *WMJJSER*, чтобы избежать совпадения сетевых сообщений со стандартными сообщениями окна. Последний параметр — *lEvent*, задает битовую маску, определяющую комбинацию сетевых событий, которые нужно отслеживать. Эти события принимают уведомления о

к *FDREAD* — готовности к чтению, \*

*FDJWRITE* — готовности к записи,

• *FDOOB* — получении срочных данных, » |я1

*FO\_ACCEPT* — входящих соединений, ж

*FDCONNECT* — завершении соединения или многоточечной операции *join*,

*FCLOSE* — закрытии сокета,

— изменении QoS,

- ***FDGROUPQOS*** — изменении QoS (зарезервировано для будущего использования группами сокетов),
- ***FD ROUTING INTERFACECHANGE*** — изменении интерфейса маршрутизации для указанных адресов,
- ***FDADDRESSLISTCHANGE*** — изменении списка локальных адресов для семейства протокола сокета

В большинстве случаев нужно отслеживать события типов *FD\_READ*, *FD\_WRITE*, *FD\_ACCEPT*, *FDJCONNECT* и *FDJLOSE*. Конечно, обработка событий *FD\_ACCEPT* или *FDJCONNECT* зависит от того, является приложение клиентом или сервером. Если приложению нужно отслеживать несколько типов событий, присвойте параметру *lEvent* значение, полученное побитовым ИЛИ над масок соответствующих типов.

```
WSAAsyncSelect(s, hWnd, WM_SOCKET,
    FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

Тогда приложение получит уведомления о событиях установления соединения, приема, передачи и закрытия сокета. При этом невозможно зарегистрировать несколько событий, происходящих на сокете одновременно. Уведомления о событиях на сокете остаются включенными, пока сокет не закрыт функцией *closesocket* или не изменен набор регистрируемых сетевых событий повторным вызовом *WSAAsyncSelect* для того же сокета. Присвоив 0 параметру *lEvent*, вы прекратите отправку всех уведомлений о событиях на сокете.

При вызове функции *WSAAsyncSelect* сокет автоматически переходит в неблокирующий режим. В результате такие функции Winsock, как *WSARecv*, при вызове возвращают ошибку *WSAEWOULDBLOCK*, если в буфере нет данных. Чтобы избежать ошибки, приложение должно опираться на пользовательское оконное сообщение, заданное в параметре *wMsg* при вызове *WSAAsyncSelect*, и показывающее, когда на сокете происходят сетевые события того или иного типа.

После успешного вызова *WSAAsyncSelect* приложение будет получать уведомления о событиях на сокете в виде сообщений Windows, отправляемых окну из параметра *hWnd*. Получающая эти сообщения процедура окна определена так:

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
```

Здесь параметр *hWnd* — описатель окна, вызвавшего оконную процедуру. Параметр *uMsg* обозначает сообщение, которое нужно обработать. В данном случае мы будем перехватывать сообщение, определенное в вызове *WSAAsyncSelect*. Параметр *wParam* определяет сокет, на котором произошло сетевое событие. Он необходим, если к одной оконной процедуре привязано несколько

сокетов. Параметр *IParam* состоит из двух частей: младшее слово указывает произошедшее событие, а старшее — содержит код ошибки.

Когда процедура окна получает сообщение о сетевом событии, она в первую очередь проверяет старшее слово параметра *IParam*, чтобы определить, не было ли ошибки. Существует специальный макрос — *WSAGETSELECTERROR*, возвращающий код ошибки в старшем слове. После этого нужно определить тип события, инициировавшего сообщение, а для этого — прочитать младшее слово *IParam*. Значение этого слова возвращает макрос *WSAGETSELECTEVENT*.

В листинге 8-5 показана обработка сообщения окна при использовании модели *WSAAsyncSelect*. Выделены последовательные шаги, необходимые для разработки любого сервера и опущены фрагменты, требующиеся для полноценной функциональности в среде Windows.

Листинг 8-5. Программирование сервера в модели *WSAAsyncSelect*

```
<define WM_SOCKET WM_USER + 1
<include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow)
{
    SOCKET Listen;
    HWND Window;

    // Создание окна и привязка процедуры ServerWinProc

    Window = CreateWmdow0;
    // Запуск Winsock и создание сокета

    WSASStartup(...);
    Listen = Socket0;
    // Привязка сокета к порту 5150 и прослушивание соединений

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sm_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);

    bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr));

    // Настройка уведомлений с сокета,
    // используя сообщение WM_SOCKET, определенное выше

    WSAAsyncSelect(Listen, Window, WM_SOCKET,
        FD.ACCEPT | FD.CLOSE);

    listenUisten, 5);
```

## Листинг 8-5. {продолжение}

// Трансляция и обработка сообщения окна до окончания работы приложения

```
BOOL CALLBACK ServerWinProc(HWND hDlg,WORD wMsg,
    WORD wParam, DWORD lParam)
{
    SOCKET Accept;

    switch(wMsg)
    {
        case WM_PAINT:
            // Обработка сообщений прорисовки окна
            break;

        case WM_SOCKET:

            // Определение возможных ошибок
            // макросом WSAGETSELECTERRORO

            if (WSAGETSELECTERROR(lParam))
            {
                // Вывод сообщения об ошибке и закрытие сокета
                closesocket(wParam);
                break;

                // Определение типа произошедшего события

                switch(WSAGETSECTEVENT(lParam))
                {
                    case FD_ACCEPT:

                        // Прием входящего соединения
                        Accept = accept(wParam, NULL, NULL);

                        // Подготовка сокета принятого соединения для отправки
                        // уведомлений о чтении, записи и закрытии

                        "Ч'.м      WSAAsyncSelect(Accept, hwnd, WM_SOCKET,
                        •У 1          FD.READ | FD_WRITE | FD_CLOSE);
                        к >Hi      break;                                     4 < W

                    ,      case FD_READ:

                        // Прием данные из сокета в wParam
                        break;

                        case FD_WRITE:
```

**Листинг 8-5.** (продолжение)

```

        // Сокет wParam готов отправлять данные
        break;

    case FD_CLOSE:

        // Закрытие соединения
        closesocket(wParam);
        break;

    break;

    return TRUE;
}

```

При обработке уведомления о событии *FDWRITE* важно следующее. Это уведомление отправляется только в одном из трех случаев:

- И** после подключения к сокету функцией *connect* win *WSAConnect*;
- Ж** после приема соединения функцией *accept* или *WSAAccept*;
- III** когда функции *send*, *WSASend*, *sendto* или *WSASendTo* возвращают ошибку *WSAEWOULDBLOCK* и место в буфере освобождается.

Поэтому приложение должно полагать, что запись в сокет возможна, начиная с первого уведомления *FDWRITE* и до тех пор, пока *send*, *WSASend*, *sendto* или *WSASendTo* не вернет ошибку *WSAEWOULDBLOCK*. После этого нужно ждать следующего уведомления *FD\_WRITE*, извещающего, что запись в сокет снова возможна.

## Модель *WSAEventSelect*

В Winsock поддерживается еще одна полезная модель асинхронного ввода-вывода, позволяющая получать уведомления о сетевых событиях на сокетах. Эта модель похожа на *WSAAsyncSelect* — приложение получает и обрабатывает те же события. Но есть и отличие — сообщения отправляются описателю объекта «событие», а не процедуре окна.

### Уведомления о событиях

Модель *WSAEventSelect* требует, чтобы приложение создало объект «событие» Для каждого сокета, вызвав функцию *WSACreateEvent*:

```
WSAEVENT WSACreateEvent(void);
```

Она возвращает описатель объекта «событие». Получив описатель объек-<sup>а</sup> нужно связать его с сокетом и зарегистрировать те типы сетевых событий, которые надо отслеживать (см. список в разделе «Модель *WSAAsyncSelect*. Уведомления о сообщениях»). Это достигается вызовом функции *WSAEventSelect*:

```

int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    int fd,
    int fd2,
    int fd3,
    int fd4,
    int fd5,
    int fd6,
    int fd7,
    int fd8,
    int fd9,
    int fd10,
    int fd11,
    int fd12,
    int fd13,
    int fd14,
    int fd15,
    int fd16,
    int fd17,
    int fd18,
    int fd19,
    int fd20,
    int fd21,
    int fd22,
    int fd23,
    int fd24,
    int fd25,
    int fd26,
    int fd27,
    int fd28,
    int fd29,
    int fd30,
    int fd31,
    int fd32,
    int fd33,
    int fd34,
    int fd35,
    int fd36,
    int fd37,
    int fd38,
    int fd39,
    int fd40,
    int fd41,
    int fd42,
    int fd43,
    int fd44,
    int fd45,
    int fd46,
    int fd47,
    int fd48,
    int fd49,
    int fd50,
    int fd51,
    int fd52,
    int fd53,
    int fd54,
    int fd55,
    int fd56,
    int fd57,
    int fd58,
    int fd59,
    int fd60,
    int fd61,
    int fd62,
    int fd63,
    int fd64,
    int fd65,
    int fd66,
    int fd67,
    int fd68,
    int fd69,
    int fd70,
    int fd71,
    int fd72,
    int fd73,
    int fd74,
    int fd75,
    int fd76,
    int fd77,
    int fd78,
    int fd79,
    int fd80,
    int fd81,
    int fd82,
    int fd83,
    int fd84,
    int fd85,
    int fd86,
    int fd87,
    int fd88,
    int fd89,
    int fd90,
    int fd91,
    int fd92,
    int fd93,
    int fd94,
    int fd95,
    int fd96,
    int fd97,
    int fd98,
    int fd99,
    int fd100,
    int fd101,
    int fd102,
    int fd103,
    int fd104,
    int fd105,
    int fd106,
    int fd107,
    int fd108,
    int fd109,
    int fd110,
    int fd111,
    int fd112,
    int fd113,
    int fd114,
    int fd115,
    int fd116,
    int fd117,
    int fd118,
    int fd119,
    int fd120,
    int fd121,
    int fd122,
    int fd123,
    int fd124,
    int fd125,
    int fd126,
    int fd127,
    int fd128,
    int fd129,
    int fd130,
    int fd131,
    int fd132,
    int fd133,
    int fd134,
    int fd135,
    int fd136,
    int fd137,
    int fd138,
    int fd139,
    int fd140,
    int fd141,
    int fd142,
    int fd143,
    int fd144,
    int fd145,
    int fd146,
    int fd147,
    int fd148,
    int fd149,
    int fd150,
    int fd151,
    int fd152,
    int fd153,
    int fd154,
    int fd155,
    int fd156,
    int fd157,
    int fd158,
    int fd159,
    int fd160,
    int fd161,
    int fd162,
    int fd163,
    int fd164,
    int fd165,
    int fd166,
    int fd167,
    int fd168,
    int fd169,
    int fd170,
    int fd171,
    int fd172,
    int fd173,
    int fd174,
    int fd175,
    int fd176,
    int fd177,
    int fd178,
    int fd179,
    int fd180,
    int fd181,
    int fd182,
    int fd183,
    int fd184,
    int fd185,
    int fd186,
    int fd187,
    int fd188,
    int fd189,
    int fd190,
    int fd191,
    int fd192,
    int fd193,
    int fd194,
    int fd195,
    int fd196,
    int fd197,
    int fd198,
    int fd199,
    int fd200,
    int fd201,
    int fd202,
    int fd203,
    int fd204,
    int fd205,
    int fd206,
    int fd207,
    int fd208,
    int fd209,
    int fd210,
    int fd211,
    int fd212,
    int fd213,
    int fd214,
    int fd215,
    int fd216,
    int fd217,
    int fd218,
    int fd219,
    int fd220,
    int fd221,
    int fd222,
    int fd223,
    int fd224,
    int fd225,
    int fd226,
    int fd227,
    int fd228,
    int fd229,
    int fd230,
    int fd231,
    int fd232,
    int fd233,
    int fd234,
    int fd235,
    int fd236,
    int fd237,
    int fd238,
    int fd239,
    int fd240,
    int fd241,
    int fd242,
    int fd243,
    int fd244,
    int fd245,
    int fd246,
    int fd247,
    int fd248,
    int fd249,
    int fd250,
    int fd251,
    int fd252,
    int fd253,
    int fd254,
    int fd255,
    int fd256,
    int fd257,
    int fd258,
    int fd259,
    int fd260,
    int fd261,
    int fd262,
    int fd263,
    int fd264,
    int fd265,
    int fd266,
    int fd267,
    int fd268,
    int fd269,
    int fd270,
    int fd271,
    int fd272,
    int fd273,
    int fd274,
    int fd275,
    int fd276,
    int fd277,
    int fd278,
    int fd279,
    int fd280,
    int fd281,
    int fd282,
    int fd283,
    int fd284,
    int fd285,
    int fd286,
    int fd287,
    int fd288,
    int fd289,
    int fd290,
    int fd291,
    int fd292,
    int fd293,
    int fd294,
    int fd295,
    int fd296,
    int fd297,
    int fd298,
    int fd299,
    int fd300,
    int fd301,
    int fd302,
    int fd303,
    int fd304,
    int fd305,
    int fd306,
    int fd307,
    int fd308,
    int fd309,
    int fd310,
    int fd311,
    int fd312,
    int fd313,
    int fd314,
    int fd315,
    int fd316,
    int fd317,
    int fd318,
    int fd319,
    int fd320,
    int fd321,
    int fd322,
    int fd323,
    int fd324,
    int fd325,
    int fd326,
    int fd327,
    int fd328,
    int fd329,
    int fd330,
    int fd331,
    int fd332,
    int fd333,
    int fd334,
    int fd335,
    int fd336,
    int fd337,
    int fd338,
    int fd339,
    int fd340,
    int fd341,
    int fd342,
    int fd343,
    int fd344,
    int fd345,
    int fd346,
    int fd347,
    int fd348,
    int fd349,
    int fd350,
    int fd351,
    int fd352,
    int fd353,
    int fd354,
    int fd355,
    int fd356,
    int fd357,
    int fd358,
    int fd359,
    int fd360,
    int fd361,
    int fd362,
    int fd363,
    int fd364,
    int fd365,
    int fd366,
    int fd367,
    int fd368,
    int fd369,
    int fd370,
    int fd371,
    int fd372,
    int fd373,
    int fd374,
    int fd375,
    int fd376,
    int fd377,
    int fd378,
    int fd379,
    int fd380,
    int fd381,
    int fd382,
    int fd383,
    int fd384,
    int fd385,
    int fd386,
    int fd387,
    int fd388,
    int fd389,
    int fd390,
    int fd391,
    int fd392,
    int fd393,
    int fd394,
    int fd395,
    int fd396,
    int fd397,
    int fd398,
    int fd399,
    int fd400,
    int fd401,
    int fd402,
    int fd403,
    int fd404,
    int fd405,
    int fd406,
    int fd407,
    int fd408,
    int fd409,
    int fd410,
    int fd411,
    int fd412,
    int fd413,
    int fd414,
    int fd415,
    int fd416,
    int fd417,
    int fd418,
    int fd419,
    int fd420,
    int fd421,
    int fd422,
    int fd423,
    int fd424,
    int fd425,
    int fd426,
    int fd427,
    int fd428,
    int fd429,
    int fd430,
    int fd431,
    int fd432,
    int fd433,
    int fd434,
    int fd435,
    int fd436,
    int fd437,
    int fd438,
    int fd439,
    int fd440,
    int fd441,
    int fd442,
    int fd443,
    int fd444,
    int fd445,
    int fd446,
    int fd447,
    int fd448,
    int fd449,
    int fd450,
    int fd451,
    int fd452,
    int fd453,
    int fd454,
    int fd455,
    int fd456,
    int fd457,
    int fd458,
    int fd459,
    int fd460,
    int fd461,
    int fd462,
    int fd463,
    int fd464,
    int fd465,
    int fd466,
    int fd467,
    int fd468,
    int fd469,
    int fd470,
    int fd471,
    int fd472,
    int fd473,
    int fd474,
    int fd475,
    int fd476,
    int fd477,
    int fd478,
    int fd479,
    int fd480,
    int fd481,
    int fd482,
    int fd483,
    int fd484,
    int fd485,
    int fd486,
    int fd487,
    int fd488,
    int fd489,
    int fd490,
    int fd491,
    int fd492,
    int fd493,
    int fd494,
    int fd495,
    int fd496,
    int fd497,
    int fd498,
    int fd499,
    int fd500,
    int fd501,
    int fd502,
    int fd503,
    int fd504,
    int fd505,
    int fd506,
    int fd507,
    int fd508,
    int fd509,
    int fd510,
    int fd511,
    int fd512,
    int fd513,
    int fd514,
    int fd515,
    int fd516,
    int fd517,
    int fd518,
    int fd519,
    int fd520,
    int fd521,
    int fd522,
    int fd523,
    int fd524,
    int fd525,
    int fd526,
    int fd527,
    int fd528,
    int fd529,
    int fd530,
    int fd531,
    int fd532,
    int fd533,
    int fd534,
    int fd535,
    int fd536,
    int fd537,
    int fd538,
    int fd539,
    int fd540,
    int fd541,
    int fd542,
    int fd543,
    int fd544,
    int fd545,
    int fd546,
    int fd547,
    int fd548,
    int fd549,
    int fd550,
    int fd551,
    int fd552,
    int fd553,
    int fd554,
    int fd555,
    int fd556,
    int fd557,
    int fd558,
    int fd559,
    int fd560,
    int fd561,
    int fd562,
    int fd563,
    int fd564,
    int fd565,
    int fd566,
    int fd567,
    int fd568,
    int fd569,
    int fd570,
    int fd571,
    int fd572,
    int fd573,
    int fd574,
    int fd575,
    int fd576,
    int fd577,
    int fd578,
    int fd579,
    int fd580,
    int fd581,
    int fd582,
    int fd583,
    int fd584,
    int fd585,
    int fd586,
    int fd587,
    int fd588,
    int fd589,
    int fd590,
    int fd591,
    int fd592,
    int fd593,
    int fd594,
    int fd595,
    int fd596,
    int fd597,
    int fd598,
    int fd599,
    int fd600,
    int fd601,
    int fd602,
    int fd603,
    int fd604,
    int fd605,
    int fd606,
    int fd607,
    int fd608,
    int fd609,
    int fd610,
    int fd611,
    int fd612,
    int fd613,
    int fd614,
    int fd615,
    int fd616,
    int fd617,
    int fd618,
    int fd619,
    int fd620,
    int fd621,
    int fd622,
    int fd623,
    int fd624,
    int fd625,
    int fd626,
    int fd627,
    int fd628,
    int fd629,
    int fd630,
    int fd631,
    int fd632,
    int fd633,
    int fd634,
    int fd635,
    int fd636,
    int fd637,
    int fd638,
    int fd639,
    int fd640,
    int fd641,
    int fd642,
    int fd643,
    int fd644,
    int fd645,
    int fd646,
    int fd647,
    int fd648,
    int fd649,
    int fd650,
    int fd651,
    int fd652,
    int fd653,
    int fd654,
    int fd655,
    int fd656,
    int fd657,
    int fd658,
    int fd659,
    int fd660,
    int fd661,
    int fd662,
    int fd663,
    int fd664,
    int fd665,
    int fd666,
    int fd667,
    int fd668,
    int fd669,
    int fd670,
    int fd671,
    int fd672,
    int fd673,
    int fd674,
    int fd675,
    int fd676,
    int fd677,
    int fd678,
    int fd679,
    int fd680,
    int fd681,
    int fd682,
    int fd683,
    int fd684,
    int fd685,
    int fd686,
    int fd687,
    int fd688,
    int fd689,
    int fd690,
    int fd691,
    int fd692,
    int fd693,
    int fd694,
    int fd695,
    int fd696,
    int fd697,
    int fd698,
    int fd699,
    int fd700,
    int fd701,
    int fd702,
    int fd703,
    int fd704,
    int fd705,
    int fd706,
    int fd707,
    int fd708,
    int fd709,
    int fd710,
    int fd711,
    int fd712,
    int fd713,
    int fd714,
    int fd715,
    int fd716,
    int fd717,
    int fd718,
    int fd719,
    int fd720,
    int fd721,
    int fd722,
    int fd723,
    int fd724,
    int fd725,
    int fd726,
    int fd727,
    int fd728,
    int fd729,
    int fd730,
    int fd731,
    int fd732,
    int fd733,
    int fd734,
    int fd735,
    int fd736,
    int fd737,
    int fd738,
    int fd739,
    int fd740,
    int fd741,
    int fd742,
    int fd743,
    int fd744,
    int fd745,
    int fd746,
    int fd747,
    int fd748,
    int fd749,
    int fd750,
    int fd751,
    int fd752,
    int fd753,
    int fd754,
    int fd755,
    int fd756,
    int fd757,
    int fd758,
    int fd759,
    int fd760,
    int fd761,
    int fd762,
    int fd763,
    int fd764,
    int fd765,
    int fd766,
    int fd767,
    int fd768,
    int fd769,
    int fd770,
    int fd771,
    int fd772,
    int fd773,
    int fd774,
    int fd775,
    int fd776,
    int fd777,
    int fd778,
    int fd779,
    int fd780,
    int fd781,
    int fd782,
    int fd783,
    int fd784,
    int fd785,
    int fd786,
    int fd787,
    int fd788,
    int fd789,
    int fd790,
    int fd791,
    int fd792,
    int fd793,
    int fd794,
    int fd795,
    int fd796,
    int fd797,
    int fd798,
    int fd799,
    int fd800,
    int fd801,
    int fd802,
    int fd803,
    int fd804,
    int fd805,
    int fd806,
    int fd807,
    int fd808,
    int fd809,
    int fd810,
    int fd811,
    int fd812,
    int fd813,
    int fd814,
    int fd815,
    int fd816,
    int fd817,
    int fd818,
    int fd819,
    int fd820,
    int fd821,
    int fd822,
    int fd823,
    int fd824,
    int fd825,
    int fd826,
    int fd827,
    int fd828,
    int fd829,
    int fd830,
    int fd831,
    int fd832,
    int fd833,
    int fd834,
    int fd835,
    int fd836,
    int fd837,
    int fd838,
    int fd839,
    int fd840,
    int fd841,
    int fd842,
    int fd843,
    int fd844,
    int fd845,
    int fd846,
    int fd847,
    int fd848,
    int fd849,
    int fd850,
    int fd851,
    int fd852,
    int fd853,
    int fd854,
    int fd855,
    int fd856,
    int fd857,
    int fd858,
    int fd859,
    int fd860,
    int fd861,
    int fd862,
    int fd863,
    int fd864,
    int fd865,
    int fd866,
    int fd867,
    int fd868,
    int fd869,
    int fd870,
    int fd871,
    int fd872,
    int fd873,
    int fd874,
    int fd875,
    int fd876,
    int fd877,
    int fd878,
    int fd879,
    int fd880,
    int fd881,
    int fd882,
    int fd883,
    int fd884,
    int fd885,
    int fd886,
    int fd887,
    int fd888,
    int fd889,
    int fd890,
    int fd891,
    int fd892,
    int fd893,
    int fd894,
    int fd895,
    int fd896,
    int fd897,
    int fd898,
    int fd899,
    int fd900,
    int fd901,
    int fd902,
    int fd903,
    int fd904,
    int fd905,
    int fd906,
    int fd907,
    int fd908,
    int fd909,
    int fd910,
    int fd911,
    int fd912,
    int fd913,
    int fd914,
    int fd915,
    int fd916,
    int fd917,
    int fd918,
    int fd919,
    int fd920,
    int fd921,
    int fd922,
    int fd923,
    int fd924,
    int fd925,
    int fd926,
    int fd927,
    int fd928,
    int fd929,
    int fd930,
    int fd931,
    int fd932,
    int fd933,
    int fd934,
    int fd935,
    int fd936,
    int fd937,
    int fd938,
    int fd939,
    int fd940,
    int fd941,
    int fd942,
    int fd943,
    int fd944,
    int fd945,
    int fd946,
    int fd947,
    int fd948,
    int fd949,
    int fd950,
    int fd951,
    int fd952,
    int fd953,
    int fd954,
    int fd955,
    int fd956,
    int fd957,
    int fd958,
    int fd959,
    int fd960,
    int fd961,
    int fd962,
    int fd963,
    int fd964,
    int fd965,
    int fd966,
    int fd967,
    int fd968,
    int fd969,
    int fd970,
    int fd971,
    int fd972,
    int fd973,
    int fd974,
    int fd975,
    int fd976,
    int fd977,
    int fd978,
    int fd979,
    int fd980,
    int fd981,
    int fd982,
    int fd983,
    int fd984,
    int fd985,
    int fd986,
    int fd987,
    int fd988,
    int fd989,
    int fd990,
    int fd991,
    int fd992,
    int fd993,
    int fd994,
    int fd995,
    int fd996,
    int fd997,
    int fd998,
    int fd999,
    int fd1000,
    int fd1001,
    int fd1002,
    int fd1003,
    int fd1004,
    int fd1005,
    int fd1006,
    int fd1007,
    int fd1008,
    int fd1009,
    int fd1010,
    int fd1011,
    int fd1012,
    int fd1013,
    int fd1014,
    int fd1015,
    int fd1016,
    int fd1017,
    int fd1018,
    int fd1019,
    int fd1020,
    int fd1021,
    int fd1022,
    int fd1023,
    int fd1024,
    int fd1025,
    int fd1026,
    int fd1027,
    int fd1028,
    int fd1029,
    int fd1030,
    int fd1031,
    int fd1032,
    int fd1033,
    int fd1034,
    int fd1035,
    int fd1036,
    int fd1037,
    int fd1038,
    int fd1039,
    int fd1040,
    int fd1041,
    int fd1042,
    int fd1043,
    int fd1044,
    int fd1045,
    int fd1046,
    int fd1047,
    int fd1048,
    int fd1049,
    int fd1050,
    int fd1051,
    int fd1052,
    int fd1053,
    int fd1054,
    int fd1055,
    int fd1056,
    int fd1057,
    int fd1058,
    int fd1059,
    int fd1060,
    int fd1061,
    int fd1062,
    int fd1063,
    int fd1064,
    int fd1065,
    int fd1066,
    int fd1067,
    int fd1068,
    int fd1069,
    int fd1070,
    int fd1071,
    int fd1072,
    int fd1073,
    int fd1074,
    int fd1075,
    int fd1076,
    int fd1077,
    int fd1078,
    int fd1079,
    int fd1080,
    int fd1081,
    int fd1082,
    int fd1083,
    int fd1084,
    int fd1085,
    int fd1086,
    int fd1087,
    int fd1088,
    int fd1089,
    int fd1090,
    int fd1091,
    int fd1092,
    int fd1093,
    int fd1094,
    int fd1095,
    int fd1096,
    int fd1097,
    int fd1098,
    int fd1099,
    int fd1100,
    int fd1101,
    int fd1102,
    int fd1103,
    int fd1104,
    int fd1105,
    int fd1106,
    int fd1107,
    int fd1108,
    int fd1109,
    int fd1110,
    int fd1111,
    int fd1112,
    int fd1113,
    int fd1114,
    int fd1115,
    int fd1116,
    int fd1117,
    int fd1118,
    int fd1119,
    int fd1120,
    int fd1121,
    int fd1122,
    int fd1123,
    int fd1124,
    int fd1125,
    int fd1126,
    int fd1127,
    int fd1128,
    int fd1129,
    int fd1130,
    int fd1131,
    int fd1132,
    int fd1133,
    int fd1134,
    int fd1135,
    int fd1136,
    int fd1137,
    int fd1138,
    int fd1139,
    int fd1140,
    int fd1141,
    int fd1142,
    int fd1143,
    int fd1144,
    int fd1145,
    int fd1146,
    int fd1147,
    int fd1148,
    int fd1149,
    int fd1150,
    int fd1151,
    int fd1152,
    int fd1153,
    int fd1154,
    int fd1155,
    int fd1156,
    int fd1157,
    int fd1158,
    int fd1159,
    int fd1160,
    int fd1161,
    int fd1162,
    int fd1163,
    int fd1164,
    int fd1165,
    int fd1166,
    int fd1167,
    int fd1168,
    int fd1169,
    int fd1170,
    int fd1171,
    int fd1172,
    int fd1173,
    int fd1174,
    int fd1175,
    int fd1176,
    int fd1177,
    int fd1178,
    int fd1179,
    int fd1180,
    int fd1181,
    int fd1182,
    int fd1183,
    int fd1184,
    int fd1185,
    int fd1186,
    int fd1187,
    int fd1188,
    int fd1189,
    int fd1190,
    int fd1191,
    int fd1192,
    int fd1193,
    int fd1194,
    int fd1195,
    int fd1196,
    int fd1197,
    int fd1198,
    int fd1199,
    int fd1200,
    int fd1201,
    int fd1202,
    int fd1203,
    int fd1204,
    int fd1205,
    int fd1206,
    int fd1207,
    int fd1208,
    int fd1209,
    int fd1210,
    int fd1211,
    int fd1212,
    int fd1213,
    int fd1214,
    int fd1215,
    int fd1216,
    int fd1217,
    int fd1218,
    int fd1219,
    int fd1220,
    int fd1221,
    int fd1222,
    int fd1223,
    int fd1224,
    int fd1225,
    int fd1226,
    int fd1227,
    int fd1228,
    int fd1229,
    int fd1230,
    int fd1231,
    int fd1232,
    int fd1233,
    int fd1234,
    int fd1235,
    int fd1236,
    int fd1237,
    int fd1238,
    int fd1239,
    int fd1240,
    int fd1241,
    int fd1242,
    int fd1243,
    int fd1244,
    int fd1245,
    int fd1246,
    int fd1247,
    int fd1248,
    int fd1249,
    int fd1250,
    int fd1251,
    int fd1252,
    int fd1253,
    int fd1254,
    int fd1255,
    int fd1256,
    int fd1257,
    int fd1258,
    int fd1259,
    int fd1260,
    int fd1261,
    int fd1262,
    int fd1263,
    int fd1264,
    int fd1265,
    int fd1266,
    int fd1267,
    int fd1268,
    int fd1269,
    int fd1270,
    int fd1271,
    int fd1272,
    int fd1273,
    int fd1274,
    int fd1275,
    int fd1276,
    int fd1277,
    int fd1278,
    int fd1279,
    int fd1280,
    int fd1281,
    int fd1282,
    int fd1283,
    int fd1284,
    int fd1285,
    int fd1286,
    int fd1287,
    int fd1288,
    int fd1289,
    int fd1290,
    int fd1291,
    int fd1292,
    int fd1293,
    int fd1294,
    int fd1295,
    int fd1296,
    int fd1297,
    int fd1298,
    int fd1299,
    int fd1300,
    int fd1301,
    int fd1302,
    int fd1303,
    int fd1304,
    int fd1305,
    int fd1306,
    int fd1307,
    int fd1308,
    int fd1309,
    int fd1310,
    int fd1311,
    int fd1312,
    int fd1313,
    int fd1314,
    int fd1315,
    int fd1316,
    int fd1317,
    int fd1318,
    int fd1319,
    int fd1320,
    int fd1321,
    int fd1322,
    int fd1323,
    int fd1324,
    int fd1325,
    int fd1326,
    int fd1327,
    int fd1328,
    int fd1329,
    int fd1330,
    int fd1331,
    int fd1332,
    int fd1333,
    int fd1334,
    int fd1335,
    int fd1336,
    int fd1337,
    int fd1338,
    int fd1339,
    int fd1340,
    int fd1341,
    int fd1342,
    int fd1343,
    int fd1344,
    int fd1345,
    int fd1346,
    int fd1347,
    int fd1348,
    int fd1349,
    int fd1350,
    int fd1351,
    int fd1352,
    int fd1353,
    int fd1354,
    int fd1355,
    int fd1356,
    int fd1357,
    int fd1358,
    int fd1359,
    int fd1360,
    int fd1361,
    int fd1362,
    int fd1363,
    int fd1364,
    int fd1365,
    int fd1366,
    int fd1367,
    int fd1368,
    int fd1369,
    int fd1370,
    int fd1371,
    int fd1372,
    int fd137
```

```
long INetworkEvents
);
```

Здесь параметр 5 — интересующий нас сокет, а параметр *hEventObject* — объект «событие», полученный из вызова *WSACreateEvent*, который нужно связать с сокетом. Последний параметр *INetworkEvents* — битовая маска получаемая комбинацией масок типов сетевых событий, которые надо отслеживать. Подробно эти типы обсуждались при описании предыдущей модели — *WSAAsyncSelect*.

У события, используемого в модели *WSAEventSelect*, два рабочих состояния — *свободное* (signaled) и *занятое* (nonsignaled), а также два оперативных режима — *ручного* (manual) и *автоматического сброса* (auto reset). Первоначально событие создается в занятом состоянии и режиме ручного сброса. Когда на сокете происходит сетевое событие, связанный с этим событием объект становится занятым. Так как объект события создается в режиме ручного сброса, приложение ответственно за его возврат в занятое состояние после обработки ввода-вывода. Это можно сделать, вызвав функцию *WSAResetEventP*.

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

Она принимает описатель события в качестве единственного параметра и возвращает *TRUE* или *FALSE*, в зависимости от успешности вызова. Закончив работу с объектом события, приложение должно вызвать функцию *WSACloseEvent* для освобождения системных ресурсов, используемых объектом:

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

Эта функция также принимает описатель события в качестве единственного параметра и возвращает *TRUE* или *FALSE*, в зависимости от успешности вызова.

Сопоставив сокет описателю события, приложение может начать обработку ввода-вывода, ожидая изменения состояния описателя. Функция *WSAWaitForMultipleEvents* будет ожидать сетевое событие и вернет управление, когда один из заданных описателей объектов событий перейдет в свободное состояние или когда истечет заданный таймаут.

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

Здесь параметры *cEvents* и *lphEvents* определяют массив объектов типа *WSAEVENT*, в котором *cEvents* — количество элементов, а *lphEvents* — указатель на массив. Функция *WSAWaitForMultipleEvents* поддерживает не более *WSA\_MAXIMUM\_WAIT\_EVENTS*(64) объектов событий. Поэтому данная модель ввода-вывода способна одновременно обслуживать максимум 64 сокета для каждого потока, вызывающего *WSAWaitForMultipleEvents*.

Если необходимо обслуживать больше сокетов, создайте дополнительные `блочные потоки` для дополнительных объектов событий. Параметр *JWaitAll* определяет, как функция *WSAWaitForMultipleEvents* реагирует на события. Если он равен *TRUE*, функция завершается после освобождения всех событий, перечисленных в массиве *iphEvents*. Если же *FALSE* — функция завершится, как только будет свободен любой объект события. В последнем случае возвращаемое значение показывает, какой именно объект был свободен.

Как правило, приложения присваивают этому параметру *FALSE* и обрабатывают одно событие сокета за раз. Параметр *dwTimeout* указывает, сколько миллисекунд функция должна ожидать сетевого события. Когда истекает таймаут, функция завершается, даже если не выполнены условия, определенные параметром *JWaitAll*. Если таймаут равен 0, функция проверяет состояние заданных объектов и выходит немедленно, что позволяет приложению эффективно проверить все события. Задавать нулевой таймаут не рекомендуется из соображений быстродействия. Если нет событий для обработки, функция *WSAWaitForMultipleEvents* возвращает значение *WSA\_WAIT\_TIMEOUT*. Если параметр *dwTimeout* равен *WSA\_INFINITE*, функция закончит работу только после освобождения какого-либо события. Последним параметром — *fAlertable*, можно пренебречь в модели *WSAEventSelect*, присвоив ему *FALSE*: он применяется в процедурах завершения процессов в модели перекрытого ввода-вывода, которая описана далее.

Функция *WSAWaitForMultipleEvents*, получив уведомление о сетевом событии, возвращает значение, определяющее его исходный объект. Найдя событие в массиве событий и связанный с ним сокет, приложение может определить, событие какого типа произошло на конкретном сокете. Для определения индекса события в массиве *IphEvents* нужно вычесть из возвращаемого значения константу *WSA\_WAIT\_EVENTJ*:

```
Index = WSAWaitForMultipleEvents(...);
MyEvent = EventArray[Index - WSA_WAIT_EVENT_0];
```

Выяснив сокет, на котором произошло событие, определяют доступные сетевые события, вызвав функцию *WSAEnumNetworkEvents*.

```
int WSAEnumNetworkEvents(
    SOCKET s,
    WSAEVENT hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents
```

Параметр *s* — сокет, на котором произошло сетевое событие. Необязательный параметр *hEventObject* — описатель связанного события, которое нужно сбросить. Так как событие в этот момент находится в свободном состоянии, можно передать его описатель для перевода в занятое состояние, <sup>не</sup>желательно использовать параметр *hEventObject*, используйте функцию *WSAResetEvent*:

```
struct _WSANETWORKEVENTS
```



```

    long INetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS

```

Последний параметр — *ipNetworkEvents*, принимает указатель на структуру *WSANETWORKEVENTS*, в которой передается тип произошедшего события и код ошибки. Параметр *INetworkEvents* определяет тип произошедшего события.

**ПРИМЕЧАНИЕ** При освобождении события иногда генерируется несколько типов сетевых событий. Например, интенсивно используемый сервер может одновременно получить сообщения *FDJREAD* и *FDJWRITE*.

Параметр *iErrorCode* — массив кодов ошибок, связанных с событиями из массива *INetworkEvents*. Для каждого типа сетевого события существует индекс события, обозначаемый тем же именем с суффиксом *BIT*. Например, для типа события *FDJREAD* идентификатор индекса в массиве *iErrorCode* обозначается *FDJREADJ3IT*. Вот анализ кода ошибки для события *FD\_READ*:

```

// Обработка уведомления FD_READ
if (NetworkEvents.INetworkEvents & FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        printf("FD_READ failed with error %d\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
    }
}

```

После обработки событий, описанных в структуре *WSANETWORKEVENTS*, приложение может продолжить ожидание сетевых событий на доступных сокетах. В листинге 8-6 показано применение модели *WSAEventSelect* для программирования сервера и управления событиями. Выделены обязательные этапы, лежащие в основе программирования сервера, способного обслуживать несколько сокетов одновременно.

#### Листинг 8-6. Пример сервера в модели ввода-вывода *WSAEventSelect*

```

SOCKET Socket[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT Event[WSA_MAXIMUM_WAIT_EVENTS];
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index;
// Настройка TCP-сокета для прослушивания порта 5150
Listen = socket (PF_INET, SOCK_STREAM, 0);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen, (PSOCKADDR) MnternetAddr,
    Sizeof(InternetAddr));

```

Листинг 8-6. (продолжение)

```
NewEvent = WSACreateEventO;

WSAEventSelect(Listen, NewEvent,
    FD_ACCEPT | FD_CLOSE);

listen(Listen, 5);

Socket[EventTotal] = Listen;
Event[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)

    // Ожидание события на всех сокетах
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);

    WSAEnumNetworkEvents(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        EventArray[Index - WSA_WAIT_EVENT_0],
        &NetworkEvents);

    // Проверка наличия сообщений FD_ACCEPT
    if (NetworkEvents.lNetworkEvents & FD_ACCEPT)

        if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0)

            printf("FD_ACCEPT failed with error Xd\n",
                NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
            break;

    // Прием нового соединения и добавление его
    // в списки сокетов и событий
    Accept = accept(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        NULL, NULL);

    // Мы не можем обрабатывать более
    // WSA_MAXIMUM_WAIT_EVENTS сокетов,
    // поэтому закрываем сокет
    if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
    {
        printf("Too many connections");
        closesocket(Accept);
        break;
    }
```

*см.след.стр.*

Листинг 8-6. *(продолжение)*

```

NewEvent = WSACreateEventO;

WSAEventSelect(Accept, NewEvent,
    FD_READ | FDWRITE | FD_CLOSE);

Event[EventTotal] = NewEvent;
Socket[EventTotal] = Accept;
EventTotal++;

printf("Socket Xd connected\n", Accept);

// Обработка уведомления FD_READ
if (NetworkEvents.lNetworkEvents & FD.READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        printf("FD_READ failed with error Xd\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
        break;

        // Чтение данных из сокета
        recv(Socket[Index - WSA_WAIT_EVENT_0],
            buffer, sizeof(buffer), 0);

// Обработка уведомления FD_WRITE
if (NetworkEvents.lNetworkEvents & FD.WRITE)
{
    if (NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
    {
        printf("FD_WRITE failed with error JSd\n",
            NetworkEvents.iErrorCode[FD_WRITE_BIT]);
        break;
    }
    send(Socket[Index - WSA_WAIT_EVENT_0],
        buffer, sizeof(buffer), 0);

if (NetworkEvents.lNetworkEvents & FD.CLOSE)
{
    if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
    {
        printf("FD_CLOSE failed with error Xd\n",
            NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
        break;
    }
}
}

```

**Листинг 8-6.** (продолжение)

```

closesocket(Socket[Index - WSA_WAIT_EVENT_0]);

// Удаление сокета и связанного события
// из массивов сокетов (Socket) и событий (Event);
// уменьшение счетчика событий EventTotal
CompressArrays(Event, Socket, &EventTotal);

```

**Модель перекрытого ввода-вывода**

Эта модель Winsock более эффективна, чем рассмотренные ранее. С ее помощью приложение может асинхронно выдать несколько запросов ввода-вывода, а затем обслужить принятые запросы после их завершения. Модель доступна на всех платформах Windows, кроме Windows CE. Схема ее работы основана на механизмах перекрытия ввода-вывода в Win32, доступных для выполнения операций ввода-вывода на устройствах с помощью функций *ReadFile* и *WriteFile*.

Первоначально модель перекрытого ввода-вывода была доступна для приложений Winsock 1.1 только в Windows NT. Приложения могли использовать эту модель, вызывая функции *ReadFile* и *WriteFile* для описателя сокета и указывая структуру перекрытия (описана далее). Начиная с версии Winsock 2, модель перекрытого ввода-вывода встроена в новые функции Winsock, такие как *WSASend* и *WSARecv*, и теперь доступна на всех платформах, где работает Winsock 2.

**ПРИМЕЧАНИЕ** Winsock 2 позволяет использовать перекрытый ввод-вывод с функциями *ReadFile* и *WriteFile* в Windows NT и 2000. Впрочем, эта функциональность не поддерживается для Windows 9x. Из соображений переносимости и производительности, применяйте функции *WSARecv* и *WSASend*, а не *ReadFile* и *WriteFile*. Мы рассмотрим лишь использование модели перекрытого ввода-вывода с новыми функциями Winsock 2.

Чтобы задействовать модель перекрытого ввода-вывода для сокета, создайте сокет с флагом *WSA\_FLAG\_OVERLAPPED*:

```

s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
  WSA_FLAG_OVERLAPPED);

```

Если вы создаете сокет функцией *socket*, а не *WSASocket*, флаг *WSA\_FLAG\_OVERLAPPED* задается неявно. Создав сокет и привязав его к локальному интерфейсу, можно использовать перекрытый ввод-вывод, вызвав следующие Функции Winsock. *WSASend*, *WSASendTo*, *WSARecv*, *WSARecvFrom*, *WSAIocctl*, *cceptEx*, *TransmitFile*. Следует также указать необязательную структуру *WSAOVERLAPPED*.

<sup>ак ВВ1</sup>, возможно, уже знаете, каждая из этих функций связана с приемом, <sup>и Пс</sup> передачей данных, или установлением соединения на соке. Их выпол-

нение потребует много времени, поэтому каждая функция может принимать структуру *WSAOVERLAPPED* в качестве параметра. Тогда они завершаются немедленно, даже если сокет работает в блокирующем режиме. В этом случае функция полагается на структуру *WSAOVERLAPPED* для завершения запроса ввода-вывода. Есть два способа завершения запросов перекрытого ввода-вывода: приложение может ожидать уведомления от объекта «событие» (event object notification) или обрабатывать завершившиеся запросы процедурами завершения (completion routines). У всех перечисленных функций (кроме *AcceptEx*) есть еще один общий параметр — *ipCompletionROUTINE*. Это необязательный указатель на процедуру завершения, которая вызывается при завершении запроса перекрытого ввода-вывода. Сначала рассмотрим способ уведомления о событиях, а затем — использование процедур завершения.

### Уведомление о событии

Для использования уведомления о событии в модели перекрытого ввода-вывода необходимо сопоставить объекты события со структурами *WSAOVERLAPPED*. Когда функции ввода-вывода, такие как *WSASend* и *WSARecv*, вызываются со структурой *WSAOVERLAPPED* в качестве параметра, они завершаются немедленно. В большинстве случаев эти вызовы возвращают ошибку *SOCKET\_ERROR*. При этом функция *WSAGetLastError* возвращает значение *WSA\_IO\_PENDING*. Это просто означает, что операция ввода-вывода продолжается. Приложению требуется определить, когда завершится перекрытый ввод-вывод. Об этом сообщает событие, связанное со структурой *WSAOVERLAPPED*. Структура *WSAOVERLAPPED* осуществляет связь между началом запроса ввода-вывода и его завершением:

```
typedef struct WSAOVERLAPPED
```

```
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
```

```
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED
```

Поля *Internal*, *InternalHigh*, *Offset* и *OffsetHigh* используются системой и не должны задаваться приложением. Поле *hEvent* позволяет приложению связать дескриптор объекта «событие» с сокетом. Этот дескриптор создают, вызвав функцию *WSACreateEvent*, как и в модели *WSAEventSelect*. После создания дескриптора события достаточно присвоить его значение полю *hEvent* структуры *WSAOVERLAPPED*, после чего можно вызывать функции Winsock, использующие структуры перекрытой модели, такие как *WSASend* или *WSARecv*.

Когда запрос перекрытого ввода-вывода завершится, приложение должно извлечь его результаты. При уведомлении посредством событий, по завершении запроса Winsock освобождает объект «событие», связанный со структурой *WSAOVERLAPPED*. Так как дескриптор этого объекта содержится в структуре *WSAOVERLAPPED*, завершение запроса перекрытого ввода-вывода легко определить, вызвав функцию *WSAWaitForMultipleEvents*, описанную в разделе посвященном модели *WSAEventSelect*.

Эта функция ждет указанное время, пока не освободится одно или несколько событий. Напомним еще раз, что функция *WSAWaitForMultipleEvents* способна одновременно обрабатывать не более 64 объектов. Выяснив, какой запрос перекрытого ввода-вывода завершился, нужно проверить, успешен ли вызов, функцией *WSAGetOverlappedResult*:

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

Параметры 5 — сокет, и *lpOverlapped* — указатель на структуру *WSAOVERLAPPED*, переданы в запросе перекрытого ввода-вывода. Параметр *ipcbTransfer* — указатель на переменную типа *DWORD*, куда записывается количество байт, фактически перемещенных операцией перекрытого ввода-вывода. Параметр *fWait* определяет, должна ли функция ждать завершения операции. Если он равен *TRUE*, функция не завершится до завершения операции, если *FALSE* и операция еще не завершилась, — функция *WSAGetOverlappedResult* вернет *FALSE* с ошибкой *WSA\_IO\_INCOMPLETE*. Так как мы ожидаем освобождения события для завершения операции ввода-вывода, этот параметр не важен. Последний параметр — *lpdwFlags*, указатель на *DWORD*, куда будут записаны результирующие флаги, если исходный перекрытый вызов осуществлялся функцией *WSARecv* или *WSARecvFrom*.

Если функция *WSAGetOverlappedResult* завершилась успешно, она возвращает *TRUE*. Это означает, что запрос перекрытого ввода-вывода успешен и значение, на которое ссылается *lpcbTransfer* обновлено. Значение *FALSE* возвращается в следующих случаях:

III операция перекрытого ввода-вывода продолжается;

- операция завершена, но с ошибками;
- \* функция *WSAGetOverlappedResult* не может определить состояние операции из-за ошибок в параметрах.

В случае неудачи значение по указателю *lpcbTransfer* не обновляется и приложение должно вызвать функцию *WSAGetLastError*, чтобы определить причины неудачи.

В листинге 8-7 показана структура простого серверного приложения, Управляющего перекрытым вводом-выводом на одном сокете с использованием уведомлений о событиях. Выделим следующие этапы.

- Создание сокета и ожидание соединения на указанном порте.
  - Прием входящего соединения.
- 3- Создание структуры *WSAOVERLAPPED* для сокета, привязка описателя объекта события к этой структуре, а также к массиву событий, который будет использоваться позднее функцией *WSAWaitForMultipleEvents*.

4. Асинхронный запрос *WSARecv* на сокет со структурой *WSAOVERLAPPED* в качестве параметра.

**ПРИМЕЧАНИЕ** Как правило, эта функция возвращает ошибку *SOCKETJERROR* со статусом *WSA\_IO\_PENDING*.

5. Вызов функции *WSAWaitForMultipleEvents* с использованием массива событий и ожидание освобождения события, связанного с запросом перекрытого ввода-вывода.
6. По завершении *WSAWaitForMultipleEvents* — сброс события функцией *WSA-ResetEvent* с массивом событий и обработка результатов запроса.
7. Определение состояния запроса перекрытого ввода-вывода функцией *WSAGetOverlappedResult*.
8. Отправка нового запроса перекрытого ввода-вывода *WSARecv* на сокет.
9. Повтор шагов 5-8.

Этот пример легко расширить для обработки нескольких сокетов: выделите в отдельный поток части кода, обрабатывающего перекрытый ввод-вывод и разрешите главному потоку приложения обслуживать дополнительные запросы соединения.

#### **Листинг 8-7. Перекрытый ввод-вывод с использованием уведомлений о событиях**

```
void main(void)
{
    WSABUF DataBuf;
    DWORD EventTotal = 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket, AcceptSocket;

    // Шаг 1:
    // Инициализация Winsock и начало прослушивания

    // Шаг 2:
    // Прием входящего соединения
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Шаг 3:
    // Формирование структуры перекрытого ввода-вывода

    EventArray[EventTotal] = WSACreateEvent0;

    ZeroMemory(&AcceptOverlapped,
        sizeof(WSAOVERLAPPED));
    AcceptOverlapped.hEvent = EventArray[EventTotal];

    DataBuf.len = DATA_BUFSIZE;
```

Листинг 8-7. *(продолжение)*

```

    DataBuf.buf = buffer;

t    EventTotal++;

    Шаг 4 :
    // Асинхронный запрос WSARecv для приема данных на сокете

    WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
            &Flags, &AcceptOverlapped, NULL);

    // Обработка перекрытых запросов на сокете

    while(TRUE)
    {
        // Шаг 5:
        // Ожидание завершения запроса перекрытого ввода-вывода
        Index = WSAWaitForMultipleEvents(EventTotal,
            EventArray, FALSE, WSA_INFINITE, FALSE);

        // Индекс должен быть равен 0, так как
        // в массиве EventArray только один описатель события

        // Шаг 6:
        // Сброс свободного события
        WSAResetEvent(
            EventArray[Index - WSA_WAIT_EVENT_0]);

        // Шаг 7:
        // Определение состояния запроса перекрытого ввода-вывода

        WSAGetOverlappedResult(AcceptSocket,
            &AcceptOverlapped, &BytesTransferred,
            FALSE, &Flags);

        // Сначала проверим, не закрыл ли
        // партнер соединение, и если да,
        // то закроем сокет

        if (BytesTransferred == 0)
        {
            printf("Closing socket %d\n", AcceptSocket);

            closesocket(AcceptSocket);
            WSACloseEvent(
                EventArray[Index - WSA_WAIT_EVENT_0]);
            return;
        }
    }

```

см. след. стр.



## Листинг 8-7. (продолжение)

```

// Обработка полученных данных,
// содержащихся в OataBuf

// Шаг 8
1 // Асинхронная отправка нового запроса WSARecvO на сокет
2
3 Flags = 0,
6 ! ZeroMemoryC&AcceptOverlapped,
    sizeof(WSAOVERLAPPED)),

AcceptOverlapped hEvent = EventArray[Index -
    WSA_WAIT_EVENT_0],

DataBuf len = DATA.BUFSIZE,
DataBuf buf = Buffer,

WSARecv(AcceptSocket, &DataBuf, 1,
    &RecvBytes, &Flags, &AcceptOverlapped,
    NULL),

```

В Windows NT и 2000 модель перекрытого ввода-вывода позволяет приложениям принимать соединения в манере перекрытия, вызывая функцию *AcceptEx* на слушающем сокете. Эта функция — специальное расширение Winsock 1.1 и доступна в файле *Mswsock.h* из библиотеки *Mswsock.lib*. Первоначально она предназначалась для перекрытого ввода-вывода в Windows NT и 2000, но работает и с Winsock 2. Функция *AcceptEx* определена так:

```

BOOL AcceptEx (
    SOCKET sListenSocket,
    SOCKET sAcceptSocket,
    PVOID IpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,           "
    DWORD dwRemoteAddressLength,         '
    LPDWORD lpdwBytesReceived,
    LPOVERLAPPED lpOverlapped
    )

```

Параметр *sListenSocket* обозначает слушающий сокет, а *sAcceptSocket* — сокет, принимающий входящее соединение. Функция *AcceptEx* отличается от *accept*: вы должны передать ей уже готовый принимающий сокет, а не создавать его функцией. Для создания сокета вызовите функцию *socket* или *WSASocket*. Параметр *IpOutputBuffer* — специальный буфер, принимающий три блока данных: локальный адрес сервера, удаленный адрес клиента и первый блок данных на новом соединении. Параметр *dwReceiveDataLength* указывает количество байт в *IpOutputBuffer*, используемых для приема данных. Если

он равен 0, при установлении соединения данные не принимаются. Параметры *dwLocalAddressLength* и *dwRemoteAddressLength* указывают, сколько байт в *lpOutputBuffer* резервируется для хранения локального и удаленного адресов при принятии соединения. Размеры буферов должны быть минимум на 16 байт больше, чем максимальная длина адреса в используемом транспортном протоколе. Например, в TCP/IP размер буфера равен размеру структуры *SOCKADDR\_IN* + 16 байт.

По адресу *ipdwBytesReceived* записывается количество принятых байт данных, если операция завершилась синхронно. Если же функция *AcceptEx* возвращает *ERROR\_IO\_PENDING*, данные по этому указателю не обновляются и количество принятых байт нужно получать через механизм уведомления о завершении. Последний параметр — *ipOverlapped*, это структура *OVERLAPPED*, позволяющая использовать *AcceptEx* для асинхронного ввода-вывода. Как упоминалось ранее, данная функция работает с уведомлениями о событиях только в приложениях с перекрытым вводом-выводом, так как в ней нет параметра процедуры завершения.

Функция *GetAcceptExSockaddrs* из расширения Winsock выделяет локальный и удаленный адреса из параметра *IpOutputBuffer*.

```
VOID GetAcceptExSockaddrs(
    PVOID IpOutputBuffer,
    DWORD dwReceiveDataLength,
    // 11  DWORD dwLocalAddressLength,
    // 12  DWORD dwRemoteAddressLength,
    // 13  LPSOCKADDR «LocalSockaddr,
    // 14  LPINT LocalSockaddrLength,
    // 15  LPSOCKADDR «RemoteSockaddr,
    // 16  LPINT RemoteSockaddrLength
);
```

Параметр *lpOutputBuffer* — указатель *lpOutputBuffer*, возвращенный функцией *AcceptEx*. Параметры *dwReceiveDataLength*, *dwLocalAddressLength* и *dwRemoteAddressLength* должны иметь те же значения, что и *dwReceiveDataLength*, *dwLocalAddressLength* и *dwRemoteAddressLength*, переданные в вызове *AcceptEx*. Параметры *LocalSockaddr* и *RemoteSockaddr* — указатели на структуры *SOCKADDR* с информацией о локальном и удаленном адресах. В них хранится смещение от адреса исходного параметра *lpOutputBuffer*. Это облегчает обращение к элементам структур *SOCKADDR*, содержащихся в *lpOutputBuffer*. Параметры *LocalSockaddrLength* и *RemoteSockaddrLength* задают размер локального и удаленного адресов.

### Процедуры завершения

Данные процедуры представляют собой еще один способ управлять завершенными запросами перекрытого ввода-вывода. По сути, это функции, которые можно передать запросу перекрытого ввода-вывода и которые вызываются системой по его завершении. Их основная роль — обслуживать завершенный запрос в потоке, откуда они были вызваны. Кроме того, приложение может продолжать обработку перекрытого ввода-вывода в процедуре завершения.

Для использования процедуры завершения приложение должно указать процедуру завершения, а также структуру *WSAOVERLAPPED* функции ввода-вывода Winsock в качестве параметра (как было описано ранее). У процедуры завершения следующий прототип:

```
void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

Когда завершается запрос перекрытого ввода-вывода, параметры функции завершения содержат следующую информацию-.

- ***dwError*** — статус завершения для перекрытой операции, на которую вызывает *lpOverlapped*;

III ***cbTransferred*** — количество байт, перемещенных при этой операции;

- ***lpOverlapped*** — структуру *WSAOVERLAPPED*, переданную в исходный вызов функции ввода-вывода;

III ***du Flags*** — не используется и равен 0.

Основное отличие перекрытого запроса с процедурой завершения от запроса с объектом события — поле *hEvent* структуры *WSAOVERLAPPED* не используется, то есть нельзя связать объект события с запросом ввода-вывода. Сделав перекрытый запрос с процедурой завершения, вызывающий поток должен обязательно выполнить процедуру завершения по окончании запроса. Для этого нужно перевести поток в *состояние ожидания* (alertable wait state) и выполнить процедуру завершения позже, по окончании операции ввода-вывода.

Для перевода потока в состояние ожидания используйте функцию *WSAWaitForMultipleEvents*. Тонкостью, что функции *WSAWaitForMultipleEvents* нужен свободный объект события. Если приложение работает только в модели перекрытого ввода-вывода с процедурами завершения, таких событий может и не быть. В качестве альтернативы можно задействовать \C^п32-функцию *SleepEx* для перевода потока в состояние ожидания. Конечно, также можно создать фиктивный, ни с чем не связанный объект события. Если вызывающий поток всегда занят и не находится в состоянии ожидания, процедура завершения никогда не будет вызвана.

Функция *WSAWaitForMultipleEvents* также может переводить поток в состояние ожидания и вызывать процедуру завершения ввода-вывода, если параметр *fAlertable* равен *TRUE*. Когда запрос ввода-вывода заканчивается через процедуру завершения, возвращается будет *WSA\_IO\_COMPLETION*, а не индекс в массиве событий. Функция *SleepEx* ведет себя так же, как и функция *WSAWaitForMultipleEvents*, кроме того, что ей не нужны объекты события:

```
DWORD SleepEx(
    DWORD dwMilliseconds,
    BOOL bAlertable
    . i i , j
```

В листинге 8-8 показана структура простого сервера, управляющего одним сокетом с использованием процедур завершения. Выделим следующие этапы.

1. Создание сокета и прослушивание соединений на заданном порте.
2. Прием входящего соединения.
3. Создание структуры *WSAOVERLAPPED* для установленного соединения.
4. Отправка асинхронного запроса *WSARecv* на сокет с заданием структуры *WSAOVERLAPPED* и процедуры завершения в качестве параметра.
5. Вызов функции *WSAWaitForMultipleEvents* с параметром *Alertable*, равным *TRUE*, и ожидание завершения запроса перекрытого ввода-вывода. Когда запрос завершается, автоматически выполняется процедура завершения и функция *WSAWaitForMultipleEvents* возвращает *WSAJO\_ JOOCompletion*. В процедуре завершения нужно отправить новый запрос перекрытого ввода-вывода *WSARecv* с процедурой завершения.
6. Проверка, что функция *WSAWaitForMultipleEvents* возвращает *WSAJO\_ JOOCompletion*.
7. Повтор шагов 5 и 6.

МШФЪИН

$$\{ \dots ; ($$

o yiodatj \ *см.след.смп-*

Листинг 8-8. *(продолжение)*

```

// обработку перекрытого ввода-вывода
// с использованием процедур завершения.
// Для этого в первую очередь
// делаем запрос WSARecv().

X
191> Flags = 0;

ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

o
DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = Buffer;
// Шаг 4:
// Отправка асинхронного запроса WSARecv() на сокет
// со структурой WSAOVERLAPPED и описанной ниже
// процедурой завершения WorkerRoutine в качестве
// параметров.

if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
    &Flags, Overlapped, WorkerRoutine)
    == SOCKET_ERROR)
{
    if (WSAGetLastError() != WSA_IO_PENDING)
    {
        printf("WSARecv() failed with error %d\n",
            WSAGetLastError());
        return;
    }

    // Так как функция WSAWaitForMultipleEvents0
    // ожидает один или несколько объектов "событие",
    // нужно создать фиктивный объект.
    // В качестве альтернативы можно использовать
    // функцию SleepEx().

    EventArray[0] = WSACreateEvent();

    while(TRUE)
    {
        // Шаг 5:
        Index = WSAWaitForMultipleEvents(1, EventArray,
            FALSE, WSA_INFINITE, TRUE);

        // Шаг 6:
        if (Index == WAIT_IO_COMPLETION)
        {
            // Процедура завершения запроса перекрытого
            // ввода-вывода закончила работу. Продолжаем
            // работу с другими процедурами завершения.

```

Листинг 8-8. (продолжение)

```

        break;

t"qWHM else
    // Произошла ошибка, нужно остановить обработку!
    // Если мы также работали с объектом "событие",
    // это может быть индекс в массиве событий,
return;

}

void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        // Или на сокете произошла ошибка,
        // или сокет закрыт партнером
        closesocket(AcceptSocket);
        return;

        // Запрос перекрытого ввода-вывода WSAREcvQ завершился
        // успешно. Теперь можно обработать принятые
        // данные, содержащиеся в DataBuf. После этого
        // нужно отправить другой запрос перекрытого ввода-вывода
        // WSAREcvO или WSA SEND(). Для простоты отправим
        // запрос WSAREcv().

        Flags = 0;

        ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

        DataBuf.len = DATA.BUFSIZE;
        DataBuf.buf = Buffer;

        if (WSAREcv(AcceptSocket, &DataBuf, 1, &RecvBytes,
                    &Flags, &Overlapped, WorkerRoutine)
            == SOCKET_ERROR)

            if (WSAGetLastError() != WSA_10_PENDING )

```

*см. след. стр.*

**Листинг 8-8.** *(продолжение)*

```

{
    printf("WSARecv() failed with error %d\n",
        WSAGetLastErrorO);
    return;

}

```

$< t$   
 $kP$

**Модель портов завершения**

Эта самая сложная модель ввода-вывода. Впрочем, она позволяет достичь наивысшего быстродействия, если приложение должно управлять многими сокетами одновременно. К сожалению, эта модель доступна только в Windows NT и 2000. Из-за сложности ее следует использовать, только если приложению необходимо управлять сотнями и тысячами сокетов одновременно и при этом нужно добиться хорошей масштабируемости при добавлении новых процессоров. Модель оптимальна только при высокоэффективном сервере под управлением Windows NT или 2000, обрабатывающем множество запросов ввода-вывода через сокеты (например, Web-сервера).

Модель портов завершения требует создать Win32-объект порта завершения, который будет управлять запросами перекрытого ввода-вывода, используя указанное количество потоков для обработки завершения этих запросов. Заметим, что на самом деле порт завершения — это конструкция ввода-вывода из Win32, Windows NT и 2000, способная работать не только с сокетами. Впрочем, здесь мы опишем лишь преимущества этой модели при работе с описателями сокетов. Для начала функцией *CreateIoCompletionPort* создадим объект порта завершения, который будет использоваться для управления запросами ввода-вывода для любого количества сокетов:

```

HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);

```

Заметьте, функция в действительности используется в двух разных целях: создания порта завершения и привязки к нему описателя сокета.

При первоначальном создании порта единственный важный параметр — *NumberOfConcurrentThreads*, первые три параметра игнорируются. Он определяет количество потоков, которые могут одновременно выполняться на порте завершения. В идеале порт должен обслуживаться только одним потоком на каждом процессоре, чтобы избежать переключений контекста. Значение 0 разрешает выделить число потоков, равное числу процессоров в системе. Создать порт завершения можно так:

```

CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
    NULL, 0, 0); sock

```

При этом возвращается описатель порта завершения, используемый при привязке сокета.

### Рабочие потоки и порты завершения

Теперь нужно связать с портом завершения описатели сокетов. Но прежде чем начинать собственно привязку, необходимо создать один или несколько рабочих потоков для обслуживания порта, когда на него отправляются запросы ввода-вывода сокетов. Но сколько именно потоков понадобится? Это весьма сложный вопрос, так как требуемое число потоков зависит от структуры приложения.

Важно понимать различие между количеством конкурентных потоков, задаваемых при вызове *CreateloCompletionPort*, и количеством создаваемых рабочих потоков — это не одно и то же. Ранее мы рекомендовали при вызове функции *CreateloCompletionPort* задавать один поток на процессор, чтобы предотвратить переключение контекста между потоками. Параметр *NumberOfConcurrentThreads* функции *CreateloCompletionPort* явно указывает системе разрешать только *n* потокам одновременно работать с портом завершения. Даже если будет создано более *n* рабочих потоков для порта завершения, только *n* смогут работать одновременно. (На самом деле, это значение может быть превышено на короткий промежуток времени, но система быстро сократит количество потоков до величины, указанной в вызове *CreateloCompletionPort*^)

Может показаться странным: зачем создавать рабочих потоков больше, чем указано в вызове *CreateloCompletionPort*? Дело в том, что если один из рабочих потоков приостанавливается (путем вызова функции *Sleep* или *WaitForSingleObject*), другой сможет работать с портом вместо него. Иными словами, всегда желательно иметь столько выполняемых потоков, сколько задано в вызове *CreateloCompletionPort*. Поэтому, если вы ожидаете, что какой-то поток будет заблокирован, лучше создать больше рабочих потоков, чем указано в параметре *CreateloCompletionPort* в вызове *NumberOfConcurrentThreads*.

Создав достаточное число рабочих потоков, начинайте собственно привязку описателей сокетов к порту. Вызовите функцию *CreateloCompletionPort* для существующего порта и передайте в первых трех параметрах (*FileHandle*, *ExistingCompletionPort* и *CompletionKey*) информацию о сокете. Параметр *FileHandle* — описатель сокета, который нужно связать с портом завершения, *ExistingCompletionPort* определяет порт завершения, а *CompletionKey* — *данные описателя* (per-handle data), которые можно связать с конкретным описателем сокета. Используя этот параметр, приложение может связать с сокетом любые данные. (Мы называем его данными описателя, потому что он представляет данные, связанные с описателем сокета.) Полезно использовать этот параметр, как указатель на структуру данных, содержащую описатель и другую информацию о сокете, чтобы процедуры потока, обслуживающие порт завершения, могли ее получать.

Теперь приступим к созданию простого приложения. В листинге 8-9 показано, как разработать приложение эхо-сервера, используя модель портов завершения. Выделим следующие этапы:



1. Создается порт завершения. Четвертый параметр равен 0, что разрешает только одному рабочему потоку на процессор одновременно выполняться на порте завершения.
2. Выясняется, сколько процессоров в системе
3. Создаются рабочие потоки для обслуживания завершившихся запросов ввода-вывода порта завершения с использованием информации о процессорах, полученной на шаге 2. В нашем простом примере мы создаем один рабочий поток на процессор, так как не ожидаем приостановки работы потоков. При вызове функции *CreateThread* нужно указать рабочую процедуру, которую поток выполняет после создания. Действия, которые в ней должен выполнить поток, мы обсудим далее.
4. Готовится сокет для прослушивания порта 5150
5. Принимается входящее соединение функцией *accept*.
6. Создается структура данных описателя и в ней сохраняется описатель принятого сокета.
7. Возвращенный функцией *accept* новый описатель сокета связывается с портом завершения вызовом функции *CreateIoCompletionPort*. Структура данных описателя передается в параметре *CompletionKey*.
8. Начинается обработка ввода-вывода на принятом соединении. Один или несколько асинхронных запросов *WSARecv* или *WSASend* отправляются на сокет с использованием механизма перекрытого ввода-вывода. Когда эти запросы завершаются, рабочий поток обслуживает их и продолжает обрабатывать новые (в рабочей процедуре на шаге 3).
- 9- Шаги 5-8 повторяются до окончания работы сервера.

#### Листинг 8-9. Настройка порта завершения

```

StartWinsock();

// Шаг 1:
// Создается порт завершения ввода-вывода

CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);

// Шаг 2:
// Выяснение количества процессоров в системе

GetSystemInfo(&SystemInfo);

// Шаг 3:
// Создание рабочих потоков для доступных процессоров в системе.
// В данном простейшем случае, мы создадим один рабочий поток
// для каждого процессора.

for(i = 0; i < SystemInfo.dwNumberOfProcessors;
```

Листинг 8-9. *(продолжение)*

```

{
    HANDLE ThreadHandle;

    // Создание рабочего потока сервера и передача
    // порта завершения в качестве параметра.
    // Примечание: процедура ServerWorkerThread
    // не определена в этом листинге.
    H
    ThreadHandle = CreateThread(NULL, 0,
    • ' ServerWorkerThread, CompletionPort,
      0, &ThreadID);
    n
    . // Закрытие описателя потока
      CloseHandle(ThreadHandle);

    // Шаг 4:
    // Создание слушающего сокета

    Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
        WSA_FLAG_OVERLAPPED);

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);
    bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr));

    // Подготовка сокета для прослушивания

    listen(Listen, 5);

    while(TRUE)
    {
        .
        // Шаг 5:
        // Прием соединений и их привязка к порту завершения

        Accept = WSAAccept(Listen, NULL, NULL, NULL, 0);

        // Шаг 6:
        // Создание структуры данных описателя,
        // которая будет связана с сокетом
        PerHandleData = (LPPER_HANDLE_DATA)
            GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

        printf("Socket number %d connected\n", Accept);
        PerHandleData->Socket = Accept;
    }
}

```

см. след. стр.

## Листинг 8-9. (продолжение)

```
// Шаг 7
// Привязка сокета к порту завершения

CreateIoCompletionPort((HANDLE) Асcept,
    CompletionPort, (DWORD) PerHandleData, 0);

// Шаг 8
// Начало обработки ввода-вывода на сокете
// Отправка одного или нескольких запросов WSASend0 или
// WSARecv() на сокет, используя перекрытый ввод-вывод
WSARecv( ),
```

## Порты завершения и перекрытый ввод-вывод

После привязки описателя сокета к порту завершения можно обрабатывать запросы ввода-вывода, отправляя сокету запросы на передачу и прием. Теперь вы вправе опираться на уведомления ввода-вывода порта завершения. Модель портов завершения использует преимущества механизма перекрытого ввода-вывода Win32, в котором вызовы функций Winsock API (таких, как *WSASend* и *WSARecv*) завершаются немедленно после вызова. Затем приложение должно правильно извлечь результаты из структуры *OVERLAPPED*. В модели портов завершения это достигается постановкой в очередь ожидания на порте завершения одного или нескольких рабочих потоков с помощью функции *GetQueuedCompletionStatus*.

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
),
```

Параметр *CompletionPort* — порт завершения, на котором будет ждать поток. Параметр *lpNumberOfBytesTransferred* принимает байты, перемещенные после операции ввода-вывода, такой как *WSASend* или *WSARecv*. В параметре *lpCompletionKey* возвращаются данные описателя сокета, который был первоначально передан в функцию *CreateIoCompletionPort*. Как мы уже упоминали, лучше передавать через этот параметр описатель сокета. В параметре *lpOverlapped* записывается перекрытый результат завершенной операции ввода-вывода. Это действительно важный параметр — он позволяет получить данные операции ввода-вывода (per I/O-operation data). Последний параметр — *dwMilliseconds*, задает, сколько миллисекунд вызывающий поток будет ждать появления пакета завершения на порте (если он равен *INFINITE*, ожидание длится бесконечно).

## Данные описателя и операции

Когда рабочий поток получает уведомление о завершении ввода-вывода от функции *GetQueuedCompletionStatus*, параметры *ipCompletionKey* и *ipOverlapped* содержат информацию о сокете, которая может быть использована для продолжения обработки ввода-вывода через порт завершения. Через эти параметры получают важные данные двух типов описателя и операции.

Параметр *ipCompletionKey* содержит данные, которые мы называем данными описателя, потому что они относятся к описателю сокета в момент первоначальной привязки к порту завершения. Эти данные передаются в параметре *CompShionKey* при вызове функции *CreateIoCompletionPort*. Как уже упоминалось, приложение может передать через этот параметр любой тип информации, связанной с сокетом. Как правило, здесь хранятся описания сокета, связанного с запросом ввода-вывода.

Параметр *IpOverlapped* содержит структуру *OVERLAPPED*, за которой следуют так называемые данные операции. В них содержатся все сведения, необходимые рабочему потоку при обработке пакета завершения (эхо-отражение данных, прием соединения, новый запрос на чтение и т.п.). Данные операции могут содержать любое количество байт вслед за структурой *OVERLAPPED*, переданной в функцию ввода-вывода, которая приняла ее в качестве параметра. Для этого проще всего определить свою структуру с первым элементом типа *OVERLAPPED*. Например, мы объявляем следующую структуру для управления данными операции.

```
typedef struct
{
    OVERLAPPED Overlapped,
    WSABUF      DataBuf,
    CHAR        Buffer[DATA_BUFSIZE],
    BOOL        OperationType,
    > PER_IO_OPERATION_DATA,
```

В эту структуру входят важные элементы данных, которые вам, может быть, придется сопоставить с операцией ввода-вывода: например тип завершившейся операции (запрос на отправку или прием). В этой структуре полезен и буфер данных для завершившейся операции. При вызове функции *Winsock*, принимающей в качестве параметра структуру *OVERLAPPED*, можно привести вашу структуру к указателю на *OVERLAPPED* или просто передать ссылку на элемент *OVERLAPPED* вашей структуры.

```
PER_IO_OPERATION_DATA PerIoData,

// функцию нужно вызывать так
    WSAREcv(socket, , (OVERLAPPED *)&PerIoData);
// или так
    WSAREcv(socket, , &(PerIoData.Overlapped)),
```

Затем, когда в рабочем потоке функция *GetQueuedCompletionStatus* вернет чл>ктуру *OVERLAPPED* (и параметр *CompShionKey*), можно определить тип запроса, который был отправлен на сокет, сняв имя с элемента структуры

*OperationType*. (Для этого приведите структуру *OVERLAPPED* к вашему типу *PER\_IO\_OPERATION\_DATA*.) Данные об операции весьма полезны, так как позволяют управлять несколькими операциями ввода-вывода (чтение-запись, множественное чтение, множественная запись и т. п.) на одном описателе. Может возникнуть вопрос: зачем отправлять запросы на несколько операций одновременно на один сокет? Для масштабируемости. Например, на многопроцессорной машине, где рабочие потоки используют все процессоры, несколько процессоров смогут отправлять и принимать данные через один сокет одновременно.

В завершение обсуждения простого эхо-сервера рассмотрим функцию *ServerWorkerThread*. В листинге 8-10 показано, как разработать процедуру рабочего потока, использующую данные операции и данные описателя для обслуживания запросов ввода-вывода.

Листинг 8-10. Рабочий поток порта завершения

```
DWORD WINAPI ServerWorkerThread(
    LPVOID CompletionPortID)
```

```
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;
```

```
    while(TRUE)
```

```
        // Ожидание завершения ввода-вывода на любом
        // из сокетов, связанных с портом завершения
```

```
        GetQueuedCompletionStatus(CompletionPort,
            SBytesTransferred, (LPDWORD)&PerHandleData,
            (LPOVERLAPPED *) &PerIoData, INFINITE);
```

```
        // Сначала проверим, не было ли ошибки
        // на сокете; если была, закрываем сокет
        // и очищаем данные описателя
        // и данные операции
```

```
        if (BytesTransferred == 0 &&
            (PerIoData->OperationType == RECVD_POSTED ||
            PerIoData->OperationType == SEND_POSTED))
```

```
            // Отсутствие перемещенных байт (BytesTransferred)
            // означает, что сокет закрыт партнером по соединению
```

```
            // и нам тоже нужно закрыть сокет. Примечание:
```

```
            // для ссылки на сокет, связанный с операцией ввода-вывода,
            // использовались данные описателя.
```

nub

"ОМ", БЭ

**Листинг 8-10.** (продолжение)

```

        closesocket(PerHandleData->Socket);

        GlobalFree(PerHandleData);
        GlobalFree(PerIoData);
        continue;

// Обслуживание завершившегося запроса ввода-вывода.
// Чтобы определить, какой запрос завершился,
// нужно просмотреть в данных операции поле OperationType.

if (PerIoData->OperationType == RECV_POSTED)
{
    // Обработка принятых данных
    // в буфере PerIoData->Buffer

    // Отправка нового запроса ввода-вывода WSASend или WSAREcv.

    // В качестве примера отправим еще один асинхронный запрос WSAREcvO

    /*      Flags = 0;

    " "      // Настройка данных операции
    ^      I/ для следующего запроса перекрытого ввода-вывода
    Г>Ъ      ZeroMemory(&(PerIoData->Overlapped),
    --<      sizeof(OVERLAPPED));
    нк      PerIoData->DataBuf.len = DATA_BUF_SIZE;
    ,st/      PerIoData->DataBuf.buf = PerIoData->Buffer;
    "      PerIoData->OperationType = RECV_POSTED;

    "      WSAREcv(PerHandleData->Socket,
    "          &(PerIoData->DataBuf), 1, &RecvBytes,
    "          &Flags, &(PerIoData->Overlapped), NULL);
    "

```

Последний момент, не отраженный ни в листингах 8-9 и 8-10, ни на прилагаемом компакт-диске — корректное закрытие порта завершения. Это особенно важно, если один или несколько выполняющихся потоков ведут ввод-вывод на нескольких сокетах. Главное — не освобождать структуру *OVERLAPPED*, пока выполняется перекрытый запрос ввода-вывода. Лучше всего вызывать функцию *closesocket* для каждого описателя сокета, тогда все операции перекрытого ввода-вывода будут завершены. После закрытия всех сокетов нужно завершить все рабочие потоки порта завершения. Отправьте каждому потоку специальный завершающий пакет функцией *PostQueuedCompletionStatus*, это заставит поток немедленно прекратить работу:

```

BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    DWORD dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);

```

Параметр *CompletionPort* — объект порта завершения, которому нужно отправить завершающий пакет. Параметры *dwNumberOfBytesTransferred*, *dwCompletionKey* и *lpOverlapped* позволяют задать значение, которое будет записано прямо в соответствующий параметр функции *GetQueuedCompletionStatus*. Когда рабочий поток получит эти три параметра, он сможет определить, когда следует прекращать работу, на основе специального значения, заданного в одном из трех параметров. Например, можно передать 0 в параметре *dwCompletionKey* — рабочий поток интерпретирует это как инструкцию об окончании работы. После закрытия всех рабочих потоков закройте порт завершения, вызвав функцию *CloseHandle*, и безопасно выйдите из программы.

### Как повысить эффективность ввода-вывода

Существует несколько приемов, позволяющих увеличить эффективность ввода-вывода через сокеты с использованием портов завершения. Один из них — опытным путем подобрать размер буфера сокета, чтобы увеличить производительность и масштабируемость приложения. Например, приложение, которое использует один большой буфер и только один запрос *WSARecv* вместо трех маленьких буферов для трех запросов *WSARecv*, будет плохо масштабироваться на многопроцессорных машинах, поскольку с одним буфером одновременно может работать только один поток. Пострадает и производительность, если одновременно выполняется только одна операция приема, драйвер сетевого протокола будет недостаточно загружен. То есть если вам приходится ждать завершения *WSARecv* перед получением новых данных, протокол будет простаивать между завершением *WSARecv* и следующим приемом.

Существует еще один способ увеличить производительность — проанализируйте результаты использования параметров сокета *SOSNDBUF* и *SO\_RCVBUF* для управления размером внутренних буферов сокета. Они позволяют приложению изменять размер внутренних буферов. Если приравнять их к 0, Winsock будет напрямую использовать буфер приложения во время перекрытого вызова для передачи данных в стек протокола и обратно, уменьшая межбуферное копирование. Следующий фрагмент показывает, как вызывать функцию *setsockopt* для настройки параметра *SOSNDBUF*.

```
int nZero = 0;
```

```

setsockopt(socket, SOL_SOCKET, SO_SNDBUF,
    (char *)&nZero, sizeof(nZero)),

```

Заметьте, нулевой размер буферов даст положительный эффект, только если несколько запросов ввода-вывода отправляются одновременно. Подробнее об этих параметрах сокета — в главе 9.

Наконец, для соединений, где передаются небольшие порции данных, производительность можно увеличить с помощью функции *AcceptEx*. Это позволяет приложению обслужить принятый запрос и извлечь данные одним вызовом API-функции, исключив издержки от отдельных вызовов функций *accept* и *WSARecv*. При этом запрос *AcceptEx* обслуживается через порт завершения, так как *AcceptEx* использует структуру *OVERLAPPED*. Функция *AcceptEx* полезна, если планируется, что сервер будет обрабатывать небольшое количество транзакций приема-передачи после установления соединения (как на Web-сервере). Если же приложение выполняет сотни и тысячи передач данных после установления соединений, реального увеличения производительности не будет.

В заключение заметим, что Winsock-приложения не должны применять \Ут32-функции *ReadFile* и *WriteFile* для обработки ввода-вывода через порт завершения. Хотя эти функции используют структуру *OVERLAPPED* и ошибки не произойдет, функции *WSARecv* и *WSASend* лучше оптимизированы для обработки ввода-вывода в Winsock 2. Обращение к *ReadFile* и *WriteFile* ведет к множеству ненужных вызовов процедур системного ядра, переключений контекста и передачи параметров, что в итоге значительно снижает производительность.

## Сравнение моделей ввода-вывода

Как же при проектировании приложения выбрать модель ввода-вывода? У каждой из них свои достоинства и недостатки, и все модели сложнее в программировании, чем простой ввод-вывод с блокировкой и несколькими потоками. Рассмотрим возможные решения для разработки клиентских и серверных приложений.

### Клиент

При разработке клиентского приложения, управляющего одним или несколькими сокетами, мы рекомендуем использовать модель перекрытого ввода-вывода или модель *WSAEventSelect* для увеличения производительности. Впрочем, при разработке приложения Windows, работающего с сообщениями окна, модель *WSAAsyncSelect* может быть лучше, так как сама опирается на модель сообщений Windows, а ваше приложение уже содержит механизм обработки сообщений.

### Сервер

При проектировании сервера, обрабатывающего несколько сокетов одновременно, рекомендуем модель перекрытого ввода-вывода. Впрочем, если вы планируете, что сервер будет одновременно обрабатывать большое количество запросов ввода-вывода, попробуйте использовать порты завершения.

## Резюме

Мы рассмотрели все модели ввода-вывода, доступные в Winsock. Они позволяют приложению использовать ввод-вывод Winsock в соответствии со сво-



ими нуждами от простого ввода-вывода с блокировкой до быстрого ввода-вывода через порт завершения

В этой главе заканчивается обсуждение общих аспектов Winsock. Мы говорили о доступных транспортных протоколах, атрибутах создания сокетов, создании простых клиент-серверных приложений и прочих фундаментальных понятиях Winsock.

В главах 9-11 будут рассмотрены специальные темы Winsock. Следующая глава посвящена параметрам сокетов и командам управления вводом-выводом, регулирующим работу как сокетов, так и базовых протоколов.

# Параметры сокета и команды управления вводом-выводом

и  
i'

Создав сокет, можно манипулировать его свойствами через параметры и команды управления вводом-выводом. Некоторые из этих параметров просто возвращают информацию, но другие — влияют на поведение сокета в приложении. Также воздействует на поведение сокета `ioctl`-команды. В этой главе обсуждаются четыре функции Winsock: *getsockopt*, *setsockopt*, *ioctlsocket* и *WSAIocctl*. У каждой множество команд, в большинстве своем плохо документированных. Мы рассмотрим обязательные и дополнительные параметры для каждой функции, а также поддерживающие их платформы. Каждый параметр предположительно работает на всех платформах Win32 (Windows CE, 95, 98, NT и 2000), если иное не оговорено. Поскольку интерфейс Winsock 2 доступен не на всех платформах, `ioctl`-команды и параметры из его состава не поддерживаются в Windows CE или Windows 95 (кроме случаев, когда для Windows 95 установлено обновление Winsock 2). Напомним, что Windows CE не поддерживает никакие параметры, не относящиеся к TCP/IP.

Большинство `ioctl`-команд и параметров описаны в файлах `Winsock.h` или `Winsock2.h` (в зависимости от их специфичности для Winsock 1 или Winsock 2). Впрочем, некоторые параметры специфичны для поставщика Microsoft или для определенного транспортного протокола. Расширения Microsoft описаны в `Winsock.h` и `Mswsock.h`. Расширения поставщиков транспорта описаны в их собственных заголовочных файлах для каждого протокола. Для таких параметров мы укажем соответствующий заголовочный файл. Приложения, использующие расширения Microsoft, нужно компоновать с библиотекой `Mswsock.lib`.

## Параметры сокета

Функция *getsockopt* наиболее часто используется для получения информации о данном сокете.

```
lit getsockopt (
    SOCKET s,
    int level,
    int optname,
    char FAR* optval,
    int FAR* optlen
)
```

Первый параметр — *s*, он определяет сокет, с которым вы будете работать. Это должен быть действительный сокет для используемого протокола. Количество параметров зависит от протокола и типа сокета, хотя некоторые применимы ко всем типам сокетов. Первый параметр связан со вторым — *level*. Параметр уровня *SOL\_SOCKET* — универсальный, не обязательно характерный для данного протокола. Мы говорим «не обязательно», потому что не все протоколы реализуют все параметры сокета на уровне *SOL\_SOCKET*. Например, *SO\_BROADCAST* переводит сокет в широковещательный режим, но не все поддерживаемые протоколы реализуют широковещательные сокеты. Фактически, нас интересует параметр *optname*.

Имена параметров — постоянные значения, определенные в заголовочных файлах *Wmsock*. Большинство общих и независимых от протокола параметров (например, с уровнем *SOL\_SOCKET*) определены в *Winsock.h* и *Winsock2.h*. У каждого протокола есть свой заголовочный файл, где определены специфичные для него параметры. И наконец, параметры *optval* и *optlen* — переменные, возвращаемые со значением интересующего вас параметра (как правило, это целое).

Функция *setsockopt* позволяет задать параметры сокета на уровне сокета или протокола:

```
int setsockopt (
    SOCKET s,
    -i int level,
    mt optname,
    ,1 const char FAR * optval,
    ,, int optlen
```

rt. Параметры те же, что и у *getsockopt*, кроме того, что вы передаете в функцию значения параметров *optval* и *optlen*, которые присваиваются определенным параметрам сокета. Как и в *getsockopt*, *optval*, как правило, целое число.

Типичная ошибка вызова *getsockopt* или *setsockopt* — попытка получить информацию о сокете, чей базовый протокол не поддерживает данную характеристику. Например, сокет типа *SOCKSTREAM* не поддерживает широковещание данных, поэтому попытка задать или получить значение параметра *SO\_BROADCAST* вызовет ошибку *WSAENOPROTOOPT*.

## Уровень *SOL\_SOCKET*

Рассмотрим параметры сокета, возвращающие информацию на основе характеристик самого сокета. Данная информация не специфична для протокола сокета.

### Параметр *SO\_ACCEPTCONN*

Этот параметр (тип *optval* — *BOOL*, версия *Winsock* 1+) можно только получить. Если возвращается *TRUE*, сокет находится в режиме прослушивания.

Если сокет был переведен в режим прослушивания функцией *listen*, возвращается *TRUE*. Сокеты типа *SOCK\_DGRAM* не поддерживают этот параметр.

## Параметр ***SO\_BROADCAST***

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сокет сконфигурирован для отправки или приема ширококешательных сообщений.

Используйте *setsockopt* с параметром *SO\_BROADCAST* для включения ширококешательных функций этого сокета. Этот параметр допустим для всех сокетов, кроме типа *SOCK\_STREAM*.

Как уже упоминалось, ширококешание — это возможность отправлять данные каждому компьютеру в локальной подсети. Конечно, на каждом компьютере должен быть запущен процесс, который прослушивает входящие ширококешательные данные. Недостаток ширококешания в том, что если множество процессов одновременно отправляют ширококешательные данные, сеть может перенасытиться, из-за чего снизится ее быстродействие. Для приема ширококешательных сообщений, активизируйте соответствующий параметр и используйте одну из функций получения дейтаграмм, например *recvfrom* или *WSARecvfrom*. Также можно подключить сокет к ширококешательному адресу, вызвав *connect* или *WSAConnect*, а затем — *recv* или *WSARecv*. Для ширококешания по UDP нужно указать номер порта для отправки дейтаграмм, аналогично, получатель должен запросить прием ширококешательных данных на том же порте. Вот пример, иллюстрирующий, как отправить ширококешательное сообщение по UDP.

```
SOCKET      s,
BOOL        bBroadcast,
char        *sMsg = "This is a test",
SOCKADDR_IN bcast,

s = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED),
bBroadcast = TRUE,
setsockopt(s, SOL_SOCKET, SO_BROADCAST, (char *)&bBroadcast,
           sizeof(BOOL)),
bcast.sin_family = AF_INET,
bcast.sin_addr.s_addr = inet_addr(INADDR_BROADCAST),
bcast.sin_port = htons(5150),
sendto(s, sMsg, strlen(sMsg), 0, (SOCKADDR *)&bcast, sizeof(bcast)),
```

В UDP предусмотрен специальный адрес, на который должны отправляться ширококешательные данные — 255.255.255.255. Ему соответствует константа *INADDR\_BROADCAST*.

AppleTalk также способен передавать ширококешательные сообщения и также предусматривает специальный адрес для их приема. В главе 6 мы упоминали, что адрес AppleTalk состоит из трех частей: сеть, узел и сокет (пункт назначения). Для ширококешания пункт назначения равен *ATADDRBROADCAST* (0xFF) — в результате дейтаграммы отправляются всем конечным точкам указанной сети.

Обычно при отправке ширококешательной дейтаграммы необходимо задать только параметр *SO\_BROADCAST*. Для приема такой дейтаграммы прослушивают только входящие дейтаграммы на заданном порте. Впрочем, при

использовании IPX в Windows 95, принимающий сокет должен задать параметр *SO\_BROADCAST* (см статью Q137914 в базе знаний по адресу <http://supportmicrosoft.com/support/search>, это ошибка Windows 95)

### Параметр *SO\_CONNECT\_TIME*

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно только получить. Он задает длительность соединения на соquete в секундах.

Параметр *SO\_CONNECT\_TIME* добавлен Microsoft. Наиболее часто он используется с функцией *AcceptEx*, которая требует, чтобы входящему клиентскому соединению был передан действительный описатель сокета. Параметр можно вызвать через клиентский описатель *SOCKET*, чтобы определить, было ли установлено соединение и сколько оно длилось. Если сокет в настоящее время не используется соединением, возвращается значение *0xFFFFFFFF*.

### Параметр *SO\_DEBUG*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить и задать. Если он равен *TRUE*, вывод отладочной информации включен.

Поставщики услуг Winsock поддерживают (но не требуют) вывод отладочной информации, если параметр *SODEBUG* задан приложением. Способ представления отладочной информации зависит от реализации базового поставщика услуг. Для включения вывода отладочной информации вызовите функцию *setsockopt* с параметром *SODEBUG* и присвойте булевой переменной *TRUE*. Вызов *getsockopt* с параметром *SOJDEBUG* возвращает *TRUE* или *FALSE*, если отладка включена или выключена, соответственно. К сожалению, ни одна из платформ Win32 в настоящее время не реализует параметр *SO\_DEBUG* (см статью Q138965 в базе знаний). При задании этого параметра ошибки не выдаются: базовый сетевой поставщик его просто игнорирует.

### Параметр *SO\_DONTLINGER*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, *SOLLINGER* отключен.

Механизм плавного закрытия сокетного соединения реализован так, что если одна или обе стороны закрывают сокет, любые данные, ожидающие в очереди или передаваемые по сети, будут отправлены или приняты обеими сторонами. Функция *setsockopt* и параметр *SOJINGER* позволяют изменить это поведение и освободить через заданный период времени сокет и все его ресурсы. Любые ожидающие или передаваемые данные, связанные с этим сокетом, отбрасываются и соединение сбрасывается (*WSAECONNRESET*). С помощью параметра *SOJDONTLINGER* можно узнать, было ли задано время задержки. Вызов *getsockopt* с параметром *SOJDONTLINGER* вернет *TRUE* или *FALSE*, если время задержки было задано или нет, соответственно. Вызов *setsockopt* с параметром *SOJDONTLINGER* отключает задержку. Сокеты типа *SOCKJGRAM* не поддерживают данный параметр.

### Параметр *SOJONTRROUTE*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сообщения отправляются прямо сетевому интерфейсу в обход таблицы маршрутов.

Параметр *SODONTRROUTE* заставляет базовый сетевой стек игнорировать таблицу маршрутов и отправлять данные напрямую интерфейсу, с которым связан сокет. Например, если вы создаете UDP-сокет и связываете его с интерфейсом А, а затем отправляете пакет, предназначенный компьютеру с интерфейсом В, пакет будет маршрутизирован так, чтобы передаваться сразу интерфейсу В. Вызов *setsockopt* с параметром *SODONTRROUTE* равным *TRUE* предотвращает маршрутизацию, в итоге пакет направляется связанному интерфейсу. Чтобы определить, включена ли маршрутизация (по умолчанию это так), вызовите функцию *getsockopt*.

Этот параметр применим на платформах Win32, однако поставщик Microsoft игнорирует такой запрос и всегда использует таблицу маршрутов, чтобы определить интерфейс для отправки исходящих данных.

### Параметр *SO\_ERROR*

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно только получить. Он возвращает статус ошибки.

Параметр *SO\_ERROR* возвращает и сбрасывает код ошибки для сокета, который отличается от кода ошибки для потока — последний обрабатывается функциями *WSAGetLastError* и *WSASetLastError*. Успешный вызов не сбрасывает код ошибки для данного сокета, возвращаемый в параметре *SO\_ERROR* — значение ошибки не всегда обновляется немедленно, поэтому есть вероятность, что в этом параметре вернется 0 (то есть, что ошибки нет). Если не требуется знать индивидуальный код ошибки, лучше всегда использовать *WSAGetLastError*.

### Параметр *SO\_EXCLUSIVEADDRUSE*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 2+) можно и получить, и задать. Если он равен *TRUE*, локальный порт, с которым связан сокет, нельзя повторно использовать в другом процессе.

Этот параметр дополняет *SO\_REUSEADDR*. Параметр *SO\_EXCLUSIVEADDRUSE* запрещает другим процессам использовать *SO\_REUSEADDR* на локальном адресе, который использует приложение. Если два отдельных процесса связаны с одним локальным адресом (учитывая, что параметр *SO\_REUSEADDR* был задан ранее), не ясно, какой из двух сокетов будет получать уведомления о входящих соединениях. Это нежелательно, особенно если приложение выполняет важную функцию. Параметр *SO\_EXCLUSIVEADDRUSE* блокирует локальный адрес, с которым связан сокет. В результате другой процесс не сможет использовать *SO\_REUSEADDR* с тем же локальным адресом. Этот параметр доступен только в Windows 2000, чтобы задать его значение, нужны полномочия администратора.

## Параметр **SO\_KEEPALIVE**

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сокет сконфигурирован для отправки в сеансе сообщений об активности соединения.

Для TCP-сокета приложение может запросить, чтобы базовый поставщик услуг передавал *пакеты сообщений об активности* (keepalive packets) TCP-соединения, включив параметр *SOKEEPALIVE*. На платформах Win32 сообщения об активности соединения реализованы, согласно разделу 4.2.3.6 из RFC 1122. После разрыва соединения любым вызовом, выполняющимся на сокете, возвращается ошибка *WSAENETRESET*. Все последующие вызовы вернут ошибку *WSAENOTCONN*. Подробности реализации этого механизма см. в RFC 1122. Важно, что сообщения об активности соединения выдаются не реже, чем через два часа — это значение задается в реестре. Изменение стандартного значения повлияет на все сообщения об активности соединения на всех TCP-соединениях в системе, что, как правило, нежелательно. Альтернативное решение — реализовать собственный механизм сообщений об активности соединения (см. также главу 7). Этот параметр не поддерживают сокет с типом *SOCKJDGRAM*.

Параметры сообщений об активности соединения регулируют параметры реестра *KeepAliveInterval* и *KeepAliveTime* (оба — типа *REGDWORD*), задающие время в миллисекундах. Первый — это интервал между повторными передачами сообщения об активности, когда не получен отклик. Второй — частота отправки пакетов для проверки доступности соединения. В Windows 95 и 98 эти параметры расположены в разделе:

```
\\KEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters
```

в Windows NT и 2000 в разделе:

```
\\KEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters
```

В Windows 2000 есть новая команда управления вводом-выводом — *SIO\_KEEPALIVE\_VALS* (описана далее), позволяющая изменить значение и интервал сообщений об активности отдельно для каждого сокета, а не в масштабе всей системы.

## Параметр **SO\_LINGER**

Этот параметр (тип *optval* — *struct linger*, версия Winsock 1+) можно и получить, и задать. С его помощью определяют текущие значения задержки.

*SO\_LINGER* задает действие, предпринимаемое после выполнения функции *closesocket*, если в очереди на сокете есть еще не отправленные данные. Вызов *getsockopt* с этим параметром возвращает текущие значения задержки в структуре *linger*.

```
struct linger {  
    u_short l_onoff;  
    u_short l_linger;
```

Ненулевое значение *l\_linger* означает, что задержка включена, а в поле *linger* хранится тайм-аут в секундах. По прошествии этого времени любые ожидающие отправки или приема данные отбрасываются, и соединение с партнером разрывается. Напротив, вы можете вызвать *setsockopt*, чтобы включить задержку и назначить тайм-аут. Для этого задайте необходимые значения в переменной с типом *struct linger*-, поле *l\_onoff* структуры не должно быть равно 0. Для отключения задержки вызовите *setsockopt* с параметром *SOJINGER*, присвоив 0 полю *l\_onoff* структуры *linger*, или *setsockopt* с параметром *SOJDONTLINGER* и передав *TRUE* в параметре *optval*. Сокеты типа *SOCK\_DGRAM* не поддерживают параметр *SODONTlinger*.

Настройка задержки напрямую влияет на поведение соединения при вызове *closesocket* (табл. 9-1).

Табл. 9-1. Параметры задержки

Параметр	Интервал	Тип закрытия	Ожидать ли закрытия
<i>SOJDONTLINGER</i>	Неприменим	Плавное	Нет
<i>SOJINGER</i>	0	Резкое	Нет
<i>SOJINGER</i>	Ненулевой	Плавное	Да

Если параметр *SOJINGER* задан с нулевым тайм-аутом (то есть поле *l\_onoff* структуры *linger* не равно 0, а поле *linger* равно 0), вызов *closesocket* не блокируется, даже если данные в очереди еще не были отправлены или подтверждены. Это называется резким или преждевременным закрытием сокета, поскольку виртуальный канал связи сбрасывается немедленно, и любые не отправленные данные теряются. Любой принимающий вызов на противоположной стороне канала вернет ошибку *WSAECNNRESET*.

Если *SOJINGER* задан с ненулевым тайм-аутом на блокирующем сокете, вызов *closesocket* блокируется на этом сокете, пока оставшиеся данные не будут отправлены или не истечет тайм-аут. Это называется плавным завершением соединения. Если таймаут истечет до отправки всех данных, реализация сокетов в Windows завершит соединение до возврата из *closesocket*.

## Параметр *SO\_MAX\_MSG\_SIZE*

Этот параметр (тип *optval* — *unsigned int*, версия Winsock 2+) предназначен только для чтения и показывает максимальный размер исходящего сообщения для типов сокетов, ориентированных на обмен сообщениями, согласно реализации конкретного поставщика службы. Он не применим к поточным сокетам. Средств определения максимального размера входящего сообщения не предусмотрено.

## Параметр *SOJDOINLINE*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, внешние данные возвращаются в обычном потоке данных.

По умолчанию *срочные данные* (out-of-band, ООБ) не передаются в основном потоке, то есть функция приема (с заданным соответствующим флагом



*MSGOOB*) возвращает OOB данные за один вызов. Если этот параметр задан, OOB-данные появляются внутри потока данных, возвращаемого вызовом приема, и чтобы определить, какой байт относится к OOB-данным, нужно вызвать *ioctlsocket* с параметром *SIOCAIMARK*. Сокеты типа *SOCKDGRAM* не поддерживают этот параметр, кроме того, он неустойчиво работает во всех текущих реализациях Win32. Подробнее об OOB-данных — в главе 7.

### Параметр **SO\_PROTOCOL\_INFO**

Этот параметр (тип *optval* — *WSAPROTOCOLINFO*, версия Winsock 2+) можно только получить. Он определяет характеристики протокола, связанного с сокетом.

Это еще один параметр только для чтения, который заполняет структуру *WSAPROTOCOL\_INFO* характеристиками протокола, сопоставленного сокету. Описание структуры *WSAPROTOCOLINFO* см. в главе 6.

### Параметр **SO\_RCVBUF**

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и получить, и задать. Он определяет размер буфера приема данных, связанного с данным сокетом. Каждому созданному сокету назначаются буферы отправки и приема. В ответ на запрос о размере буфера приема в вызове *setsockopt* не будет выдана ошибка, даже если реализация не в состоянии предоставить буфер указанного размера. Поэтому чтобы узнать, какой буфер фактически выделен, вызовите *getsockopt*. Задавать и получать размер буфера приема позволяют все платформы Win32, кроме Windows CE, где можно лишь узнать размер этого буфера.

Вам может потребоваться изменить размер буфера, чтобы приспособить его к своему приложению. Например, в коде, принимающем UDP-дейтаграммы, размер буфера приема, как правило, должен быть кратен размеру дейтаграммы.

При перекрытом вводе-выводе нулевой размер буфера может увеличить производительность, например, если требуется дополнительно копировать содержимое памяти для переноса данных из системного буфера в пользовательский. В отсутствие промежуточного буфера данные копируются сразу в пользовательский. Впрочем, такой подход эффективен, только если нужно обрабатывать множество ожидающих вызовов приема. При асинхронной обработке единичных операций приема производительность снизится, поскольку локальная система не сможет принимать входящие данные, пока у вас не будет для них готового буфера. Подробнее об этом — в главе 8.

### Параметр **SO\_REUSEADDR**

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, вы вправе связать сокет только с адресом, уже используемым другим сокетом, или с адресом в состоянии *TIME\_WAIT*.

По умолчанию сокет не может быть связан с уже используемым локальным адресом, однако иногда необходимо многократно использовать адрес

таким образом Мы уже говорили, что каждое соединение однозначно идентифицируется комбинацией его локальной и удаленного адресов Если адрес, с которым вы соединяетесь, хоть чем-то уникален (например, у него другой номер порта TCP/IP), привязка к нему разрешена

Единственное исключение — слушающие сокеты Два разных сокета нельзя связать с одним локальным интерфейсом (и портом — в случае TCP/IP) для ожидания входящих соединений Когда два сокета активно прослушивают на одном порте, невозможно предсказать, какой именно из них получит уведомление о входящем соединении Параметр *SO\_REUSEADDR* наиболее полезен в TCP, если сервер завершил работу (в том числе преждевременно), оставив локальный адрес и порт в состоянии *TIME\_WAIT* Тогда другие сокеты будет невозможно связать с «зависшим» портом Если задать этот параметр, сервер сможет прослушивать на том же локальном интерфейсе и порте после перезагрузки

### Параметр *SO\_SNDBUF*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать Он сравнительно прост и определяет размер буфера отправки данных, связанного с конкретным сокетом Значение *TRUE* (ненулевое) означает, что сокет настроен для отправки широкоэмительных сообщений Каждому созданному сокету назначаются буферы отправки и приема В ответ на запрос о размере буфера приема в вызове *setsockopt* не будет выдана ошибка, даже если реализация не в состоянии предоставить буфер указанного размера Поэтому чтобы узнать какой буфер фактически выделен, вызовите *getsockopt* Задавать и получать размер буфера отправки позволяют все тат-формы Win32, кроме Windows CE, где его можно лишь узнать

Как и *SO\_JCVBUF*, параметр *SO\_SNDBUF* можно использовать, чтобы задать нулевой размер буфера отправки — тогда по завершении блокирующего вызова отправки можно быть уверенным, что данные переданы в сеть Кроме того, как и в операции приема без буфера, не требуется дополнительно копировать содержимое памяти в системные буферы Однако при этом вы теряете преимущества конвейерной работы буферов стека, когда их размер не равен 0 Другими словами, если программа циклически отправляет данные, локальный сетевой стек копирует эти данные в системный буфер, который фактически отправляет их по мере возможности (зависит от используемой модели ввода-вывода)

Если же логика вашего приложения иная, отключение буферов отправки сэкономит вам несколько машинных команд на копировании памяти Подробнее об этом — в главе 8

### Параметр *SO\_JTYPE*

Этот параметр *SO\_JTYPE* (тип *optval* — *mt*, версия Winsock 1+) можно только получить Он доступен только для чтения и просто возвращает тип данного сокета *SOCK\_JDGRAM*, *SOCK\_JTREAM*, *SOCK\_JEQPACKET*, *SOCK\_JRDM* и *SOCK\_JWW*

### Параметр **SO\_SNDTIMEO**

Этот параметр (тип *opWal* — *int*, версия Winsock 1+) можно и задать, и получить. Он определяет тайм-аут на блокирующем сокете, учитываемый при вызовах функций отправки данных. Значение тайм-аута в миллисекундах указывает длительность блокирования при отправке данных. Если необходимо использовать *SO\_SNDTIMEO* вместе с функцией *WSASocket* (для создания сокета), укажите *WSA\_FLAG\_OVERLAPPED* в составе параметра *divFlags* функции *WSASocket*. Последующие вызовы любой Winsock-функции отправки (*send*, *sendto*, *WSASend*, *WSASendTo* и т. п.) блокируют только на заданную величину времени. Если операция отправки не завершается за это время, вызов вернет ошибку 10060 (*WSAETIMEDOUT*).

Для увеличения производительности этот параметр отключен в Windows CE 2.1 — он просто игнорируется, ошибка не выдается. В предыдущих версиях Windows CE параметр работает.

### Параметр **SO\_RCVTIMEO**

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и задать, и получить. Он назначает тайм-аут на блокирующем сокете, учитываемый при вызовах функций приема данных. Значение таймаута в миллисекундах определяет длительность блокирования при приеме данных. Если необходимо использовать *SO\_SNDTIMEO* вместе с функцией *WSASocket* (для создания сокета), укажите *WSA\_FLAG\_OVERLAPPED* в составе параметра *divFlags* функции *WSASocket*. Последующие вызовы любой Winsock-функции приема (*recv*, *recvfrom*, *WSARecv*, *WSARecvFrom* и т. п.) блокируют только на заданную величину времени. Если операция отправки не завершается за это время, вызов вернет ошибку 10060 (*WSAETIMEDOUT*).

Для увеличения производительности этот параметр отключен в Windows CE 2.1 — он просто игнорируется, ошибка не выдается. В предыдущих версиях Windows CE параметр работает.

### Параметр **SO\_UPDATE\_ACCEPT\_CONTEXT**

Этот параметр (тип *optval* — *SOCKET*, версия Winsock 1+) можно и задать, и получить. Он определяет таймаут приема данных на сокете (в миллисекундах). Это расширение Microsoft чаще всего используется вместе с функцией *AcceptEx*. Уникальная особенность данной функции — она входит в спецификацию Winsock 1 и позволяет использовать перекрытый ввод-вывод для обработки вызова приема. Функции *AcceptEx* в качестве параметра передается прослушивающий сокет, а также описатель сокета, который должен быть принят клиентом.

С помощью параметра *SO\_UPDATE\_ACCEPT\_CONTEXT* клиентскому сокету назначают характеристики прослушивающего сокета. Этот параметр обязателен для прослушивающего сокета QoS-приложения — в функции *setsockopt* используйте прослушивающий сокет как параметр *SOCKET*, а описатель принимающего сокета (например, клиента) — как параметр *optval*. Параметр *SO\_UPDATE\_ACCEPT\_CONTEXT* применяется лишь в Windows NT и 2000.

## Уровень параметров *SOL\_APPLETALK*

Описанные в этом разделе параметры специфичны для протокола AppleTalk и могут использоваться только с сокетами, созданными функцией *socket* или *WSASOCKETs* флагом *AF\_APPLETALK*. Большинство этих параметров связано с заданием либо с получением имен AppleTalk. Подробнее о семействе адресов AppleTalk — в главе 6. У некоторых параметров сокета AppleTalk (типа *SO\_DEREGISTER\_NAME*) несколько имен, все они взаимозаменяемы.

### Параметр *SO\_CONFIRM\_NAME*

Этот параметр (тип *optval* — *WSH\_NBP\_TUPLE*, версия Winsock 1) можно только получить. Он подтверждает, что данное имя AppleTalk связано с указанным адресом. Для проверки имени по этому адресу отправляется запрос поиска по протоколу Name Binding Protocol (NBP). Если он возвращает ошибку *WSAEADDRNOTAVAIL*, значит имя больше не связано с адресом.

### Параметры *SO\_DEREGISTER\_NAME* и *SO\_REMOVE\_NAME*

Этот параметр (тип *optval* — *WSH\_REGISTER\_NAME*, версия Winsock 1) можно и получить, и задать. Они отменяют регистрацию имени в сети. Если на данный момент имени в сети не существует, вызов выполнится успешно. Подробнее о структуре *WSH\_REGISTER\_NAME*, которая, по сути, просто другое название *WSH\_NBP\_NAME* — в главе 6.

### Параметры *SO\_LOOKUP\_MYZONE* и *SO\_GETMYZONE*

Эти параметры (тип *optval* — *char'*, версия Winsock 1) можно только задать. Они возвращают стандартную зону сети. Параметр *optval* для *getsockopt* — строка, длиной минимум 33 символа. Как вы знаете, максимальная длина NBP-имени — *MAX\_ENTHLY\_LEN*, который равен 32. Дополнительный символ необходим для нулевого терминатора строки.

### Параметр *SO\_LOOKUP\_NAME*

Этот параметр (тип *optval* — *WSH\_LOOKUP\_NAME*, версия Winsock 1) можно только задать. Его применяют для поиска в сети указанного имени NBP, он возвращает соответствующие комбинации имен и информации NBP (например, когда клиент хочет соединиться с сервером). Перед установлением связи нужно разрешить в адрес AppleTalk *стандартное* (well-known) текстовое имя. Пример кода для поиска имени AppleTalk — в главе 6.

Учтите, что после успешного возвращения структуры *WSH\_NBP\_TUPLE* располагаются в буфере вслед за информацией *WSH\_LOOKUP\_NAME*. Поэтому нужно указать в вызове *getsockopt* буфер достаточного размера, чтобы туда поместились все возвращенные сведения. *WSH\_LOOKUP\_NAME* — в начале буфера и ряды структур *WSH\_NBP\_TUPLE* — в конце (рис. 9-1)-

Рис. 9-1. Буфер *SOJLOOKUPNAME*

## Параметры *SO\_LOOKUP\_ZONES* и *SO\_GETZONELIST*

Этот параметр (тип *optval* — *WSH\_LOOKUP\_ZONES*, версия Winsock 1) можно только получить. Он возвращает имена зон возвращений из списка зон в Интернете. Нужен достаточно вместительный буфер, чтобы разместить в его начале структуру *WSH\_LOOKUP\_ZONES*. Если возвращение успешно, пространство после структуры *WSH\_LOOKUP\_ZONES* содержит список имен зон/с нулевым символом в конце. Приведенный далее код иллюстрирует использование параметра *SO\_LOOKUP\_ZONES*. I

```
PWSH_LOOKUP_NAME    atlookup,
PWSH_LOOKUP_ZONES   zonelookup;
char                 cLookupBuffer[4096],
                    •pTupleBuffer = NULL;


atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
ret = getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES, (char *)atlookup,
                &dwSize);
pTupleBuffer = (char *)cLookupBuffer + sizeof(WSH_LOOKUP_ZONES);
for(i = 0; 1 < zonelookup->NoZones;

    prmtf("X3d. 'Xs'\n", 1 + 1, pTupleBuffer);
    while (*pTupleBuffer++);
```

## Параметры *SO\_LOOKUP\_ZONES\_ON\_ADAPTER* и *SO\_GETLOCALZONES*

Эти параметры (тип *optval* — *WSH\_LOOKUP\_ZONES*, версия Winsock 1) можно только получить. Они возвращают список имен зон, известных адаптеру с указанным именем. Это похоже на работу *SO\_LOOKUP\_ZONES*, за одним исключением: при использовании *SO\_LOOKUP\_ZONES\_ON\_ADAPTER* и *SO\_GETLOCALZONES* вы можете указать имя адаптера и получить список зон для локальной сети, к которой тот подключен. Также необходим вместительный буфер, в начале которого разместится структура *WSH\_LOOKUP\_ZONES*. Возвращенный список имен зон, разделенных нулевым символом, начинается после структуры *WSH\_LOOKUP\_ZONES*. Имя адаптера должно быть строкой UNICODE (*WCHAR*).

## Параметры *SO\_LOOKUP\_NETDEF\_ON\_ADAPTER* и *SO\_GETNET1INFO*

Эти параметры (тип *optval* — *WSH\_LOOKUP\_NETDEF\_ON\_ADAPTER*, версия Winsock 1) можно только задать. Они возвращают диапазоны сетевых номеров и ANSI-строку с нулевым символом в конце, содержащую стандартную зону для сети на указанном адаптере. Сведения об адаптере передаются в строке UNICODE (*WCHAR*) вслед за структурой и перезаписываются информацией о стандартной зоне по возвращении. Если сеть не *фрагментирована* (seeded), возвращается сетевой диапазон 1-0xFFFFE, а в ANSI-строке с нулевым символом в конце содержится стандартная зона — 

### Параметр **SO\_PAP\_GET\_SERVER\_STATUS**

Этот параметр (тип *optval* — *WSH\_PAP\_GET\_SERVER\_STATUS*, версия Winsock 1) можно только получить. Он возвращает статус PAP от заданного сервера — отображает состояние протокола Printer Access Protocol (PAP), зарегистрированного по адресу, указанному в *ServerAddr* (обычно известен по поиску NBP). Четыре зарезервированных байта соответствуют четырем зарезервированным байтам в пакете состояния PAP (в сетевой порядке байт). Строка состояния PAP не стандартизована и задается параметром *SO\_PAP\_SET\_SERVER\_STATUS*. Структура *WSH\_PAP\_GET\_SERVER\_STATUS* определена так.

```
#define    MAX_PAP_STATUS_SIZE    255
#define    PAP_UNUSED_STATUS_BYTES    4

typedef struct _WSH_PAP_GET_SERVER_STATUS
{
    SOCKADDR_AT    ServerAddr;
    USHORT    Reserved[PAP_UNUSED_STATUS_BYTES];
    USHORT    ServerStatus[MAX_PAP_STATUS_SIZE + 1];
} WSH_PAP_GET_SERVER_STATUS, *PWSH_PAP_GET_SERVER_STATUS;
```

Приведем краткий пример опроса состояния PAP. Длина строки состояния записана в первом байте поля *ServerStatus*.

```
WSH_PAP_GET_SERVER_STATUS    status;
int    nSize = sizeof(status);

status.ServerAddr.sat_family = AF_APPLETALK;
ret = getsockopt(s, SOL_APPLETALK, SO_PAP_GET_SERVER_STATUS,
    (char *)&status, &nSize);
```

### Параметр **SO\_PAP\_PRIME\_READ**

Этот параметр (тип *optval* — *char[]*, версия Winsock 1) можно только задать. Его вызов предваряет чтение по соединению PAP. На сокете, описывающем подключение PAP, он позволяет удаленному клиенту отправить данные, даже если локальное приложение не вызвало *recv* или *WSARECVEX*. Задав этот параметр, приложение может блокировать вызов *select*, а затем считать данные. Параметр *optval* для этого вызова — буфер, принимающий данные, длиной минимум *MIN\_PAP\_READ3UF\_SIZE* (4096) байт. Он позволяет работать с неблокирующими сокетами по протоколу PAP. Заметьте, что для каждого буфера, который вы хотите считать, необходимо вызвать *setsockopt* с параметром *SO\_PAP\_PRIME\_READ*.

### Параметр **SO\_PAP\_SET\_SERVER\_STATUS**

Этот параметр (тип *optval* — *char[]*, версия Winsock 1) можно только задать. Он определяет состояние PAP, отправляемое в ответ на соответствующий запрос другого клиента. При этом клиенту будет возвращаться содержимое

указанного буфера, объем которого — не более 255 байт. Если задать пустой буфер состояния, его предыдущее содержимое будет стерто.

### Параметр *SO\_REGISTER\_NAME*

Этот параметр (тип *optval* — *WSH\_REGISTER\_JVMIE*, версия Winsock 1) можно только задать. Он используется для регистрации указанного имени в сети AppleTalk. Если это имя уже существует в сети, выдается ошибка *WSAEAD\*DRINUSE*. Подробнее о структуре *WSH\_REGISTER\_NAME* — в главе 6.

## Уровень параметров *SOLJRLMP*

Уровень *SOLJRLMP* связан с протоколом IrDA (семейство адресов *AFJIRDA*). При использовании описанных далее параметров учтите, что инфракрасные (ИК) сокеты реализованы на всех платформах по-разному. Поскольку Windows CE первая стала поддерживать IR, в этой ОС доступны не все параметры, представленные позже в Windows 98 и Windows 2000. В этом разделе описание каждого параметра мы сопровождаем перечнем поддерживающих его платформ.

### Параметр *IRLMP\_9WIRE\_MODE*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Он определяет параметры IP в рамках заголовка IP. Это еще один редко используемый параметр, необходимый для связи с Windows 98 по IrCOMM, уровень которого ниже, чем тот, на котором обычно работает IrSock. В девятипроводном режиме каждый пакет TinyTP или IrLMP содержит дополнительный однобайтовый заголовок IrCOMM.

Чтобы добиться этого через интерфейс сокета, необходимо сначала получить максимальный размер PDU для IrLMP-пакета с параметром *IRLMP\_SEND\_PDU\_LEN*. Затем перед началом или приемом соединения сокет переводится функцией *setsockopt* в девятипроводной режим. Она дает указание стеку добавлять однобайтовый заголовок IrCOMM (всегда равный 0) к каждому отправляемому кадру. Поэтому размер каждого отправляемого *send* блока данных должен быть меньше максимальной длины PDU, дабы оставить место для добавленного байта IrCOMM.

Протокол IrCOMM не рассматривается в этой книге. Параметр доступен на Windows 98 и Windows 2000.

### Параметр *IRLMP^ENUMDEVICES*

Этот параметр (тип *optval* — *DEVICELIST*, версия Winsock 1+), можно только получить. Инфракрасные устройства связи по своей природе мобильны и могут входить и выходить из области видимости. Этот параметр «опрашивает» ИК-устройства в области видимости, нумеруя их, и возвращает список их идентификаторов.

Структуры *DEVICELIST* на разных платформах, поддерживающих IrSock, отличаются друг от друга, поскольку новые платформы обладают расширенной функциональностью. Изначально поддержка IrSock была реализована в

Windows CE, а в Windows 98 и Windows 2000 появилась позже. Определение структуры *DEVICELIST* для Windows 98 и Windows 2000 выглядит так:

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                numDevice;
    WINDOWS_IRDA_DEVICE_INFO Device[1];
} WINDOWS_DEVICELIST, *PWINDOOWS_DEVICELIST, FAR *LPWINDOOWS_DEVICELIST;

typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char irdaDeviceID[4];
    /* char irdaDeviceName[22];
    ** u_char irdaDeviceHints1;
    * u_char irdaDeviceHints2;
    u_char irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOOWS_IRDA_DEVICE_INFO,
FAR *LPWINDOOWS_IRDA_DEVICE_INFO;
```

В Windows CE структура *DEVICELIST* определена так:

```
typedef struct _WCE_DEVICELIST
{
    ULONG                numDevice;
    WCE_IRDA_DEVICE_INFO Device[1];
} WCE_DEVICELIST, *PWCE_DEVICELIST;

typedef struct _WCE_IRDA_DEVICE_INFO
{
    u_char irdaDeviceID[4];
    char irdaDeviceName[22];
    u_char Reserved[2];
} > WCE_IRDA_DEVICE_INFO, *PWCE_IRDA_DEVICE_INFO;
```

Как видите, структура информации устройства тоже иная: *WCE\_IRDA\_DEVICE\_INFO* — для Windows CE и *WINDOWS\_IRDA\_DEVICE\_INFO* — для Windows 98 и Windows 2000. Каждая из этих структур содержит поле *irdaDeviceID* — четырехбайтный тег, однозначно идентифицирующий устройство. Это поле необходимо для заполнения структуры *SOCKADDR\_IRDA*, которую используют для соединения с конкретным устройством, а также для получения записи *службы доступа к информации* (Information Access Service, IAS) с параметрами *IRLMP\_IASJET* и *IRLMP\_IAS\_QUERY*.

При перечислении ИК-устройств в ходе вызова *getsockopt* необходимо, чтобы параметр *optval* был структурой *DEVICELIST*. Единственное требование — сначала присвоить 0 полю *numDevice*. Вызов *getsockopt* не возвращает ошибку, если ИК-устройства не обнаружены. Необходимо проверить значение поля *numDevice*: если оно больше 0, то было найдено одно или несколько устройств. Поле *Device* возвращается с числом структур, равным значению в поле *numDevice*.



### Параметр **IRLMP\_EXCLUSIVE\_MODE**

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сокетное соединение работает в монопольном режиме. Параметр обычно не употребляется пользовательскими приложениями, поскольку обходит уровень TinyTP в стеке IrDA и связывается напрямую с IrLMP. Если вы действительно заинтересованы в его использовании, ознакомьтесь со спецификацией IrDA по адресу <http://www.irda.org>. Доступен в Windows CE и Windows 2000.

,  
f

### Параметр **IRLMPJASQUERY**

Этот параметр (тип *optval* — *IAS\_QUERY*, версия Winsock 1+) можно только получить. Он опрашивает IAS относительно атрибутов указанной службы и класса. Параметр дополняет *IRLMPJASSET* и находит информацию об имени класса и его службе.

Перед вызовом *getsockopt* сначала заполните поле *irdaDeviceID* для ссылки на опрашиваемое устройство. Введите в поле *irdaAttnbName* строку свойства, по которой хотите отыскать его значение. Как правило, опрашивается номер LSAP-SEL, его строка свойства — IrDA IrLMP LsapSel. Затем задайте полю *irdaClassName* имя службы, которой соответствует данная строка свойства. После заполнения этих полей вызовите *getsockopt*. В случае успеха *irdaAttnbType* указывает, из какого поля объединения следует получать информацию. Для декодирования этой записи используйте идентификаторы из табл. 9-2 (вы найдете ее чуть далее в этой главе). Типичная ошибка *WSASERVICE\_NOT\_FOUND* возвращается, если данная служба не обнаружена на указанном устройстве. Параметр доступен в Windows CE/98/2000.

### Параметр **IRLMPJASSET**

Этот параметр (тип *optval* — *IAS\_QUERY*, версия Winsock 1+) можно только задать. Служба IAS — это управляемый механизм динамической регистрации. *IRLMPJASSET* позволяет задать один атрибут для одного класса в локальной IAS. Как и в случае с *IRLMP\_ENUMDEVICES*, предусмотрены разные структуры для Windows CE/98/2000. Структуры для Windows 98 и Windows 2000 таковы:

```
typedef struct _WINDOWS_IAS_QUERY
```

```

    u_char    irdaDeviceID[4],
    char      irdaClassName[IAS_MAX_CLASSNAME];
    char      irdaAttnbName[IAS_MAX_AnRIBNAME]
    u_long    irdaAttnbType,
    union
    {
        LONG    irdaAttnbInt,
        struct
        {
            u_long    Len,
            u_char    OctetSeq[IAS_MAX_OCTET_STRING];
        } irdaAttribOctetSeq,
    }

```

```

struct
    u_long    Len,
    u_long    CharSet,
    u_char    UstrStr[IAS_MAX_USER_STRING],
} IrdaAttnbUstrStr,
} IrdaAttribute,
} WINDOWS_IAS_QUERY, *PWINDOVS_IAS_QUERY, FAR *LPWINDOVS_IAS_QUERY,

```

Структура запроса IAS для Windows CE

```

typedef struct _WCE_IAS_QUERY
{
    u_char    irdaDeviceID[4],
    j char    irdaClassName[61],
    char      irdaAttnbName[61],
    u_short   lrdaAttribType,
    union
    {
        h int    lrdaAttnblnt,
        tf struct
        {
            jy
            *
            int    Len,
            u_char  OctetSeq[1],
            u_char  Reserved[3],
            } lrdaAttribOctetSeq,
            struct
            {
                int    Len,
                u_char  CharSet,
                u_char  UstrStr[1],
                u_char  Reserved[2],
                1
            } lrdaAttnbUstrStr,
            } irdaAttribute,
    } WCE_IAS_QUERY *PWCE_IAS_QUERY,

```

В табл. 9-2 перечислены разные константы для поля *irdaAttnbType*, которое указываеа тип атрибута. Значения для двух последних записей включены в таблицу лишь для полноты информации. Задать эти значения нельзя: их может вернуть в поле *lrdaAttnbtype* вызов *getsockopt* с параметром *IRIMPJAS\_QUERY*.

Табл. 9-2. Типы атрибута IAS

Значение <i>irdaAttnbType</i>	Поле
<i>IAS_ATTRIB_INT</i>	<i>lrdaAttnblnt</i>
<i>IAS_ATTRIBOCTETSEQ</i>	<i>lrdaAttribOctetSeq</i>
<i>IAS_ATTRIBJTR</i>	<i>lrdaAttribUvStr</i>
<i>IAS_ATTRIB_NO_CLASS</i>	Нет
<i>IAS_ATTRIB_NO_ATTRIB</i>	Нет

Чтобы задать значение, укажите в поле *u daDeviceID* ИК-устройство, для которого изменяется запись IAS. Кроме того, в поле *irdaAttnbName* следует задать класс, а в *irdaClassName* — службу для атрибута. Помните, что при использовании IrSock серверы сокета — это службы, зарегистрированные средствами IAS, которым сопоставлен номер LSAP-SEL. Данный номер клиенты используют для соединения с сервером.

Для изменения номера LSAP-SEL входа службы IAS укажите в поле *irda-DeviceID* идентификатор устройства, на котором запущена служба. В поле *irdaAttnbName* запишите строку IrDA IrLMP LsapSel, а в поле *irdaClassName* — имя службы (например, MySocketServer). Затем присвойте значение *IAS\_ATTRIBJNT* полю *irdaAttnbType*, а в *irdaAttnblnt* укажите новый номер LSAP-SEL. Конечно, номер LSAP-SEL лучше не изменять. Этот пример дан здесь лишь для наглядности.

### Параметр **IRLMPJRLPT\_MODE**

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сокет настроен для связи с ИК-принтерами. Средствами Winsock можно подключиться к инфракрасному принтеру и отправить задание печати. Для этого перед установлением связи переведите сокет в режим IRLPT. Просто присвойте этому параметру значение *TRUE* после создания сокета.

Чтобы найти в пределах диапазона принтеры, поддерживающие инфракрасный режим, используйте параметр *IRLMP\_ENUMDEVICES*. Некоторые устаревшие ИК-принтеры не регистрируют себя в IAS, возможно, вам придется подключиться к ним напрямую по идентификатору <LSAP-SEL-XYX> (подробнее о способах обхода IAS — в главе 6). Этот параметр доступен в Windows CE/2000.

### Параметр **RLMP\_SEND\_PDU\_LEN**

Этот параметр (тип *optval* — *mt*, версия Winsock 1+) можно только получить. Он отображает максимальный размер блока данных протокола (Protocol Data Unit, PDU), необходимый при использовании параметра *IRLMP\_9WIRE\_MODE*. Более подробно этот параметр, доступный в Windows CE/2000, рассмотрен в описании *IRLMPJWIREJ4ODE*.

## Уровень параметров **IPPROTOJP**

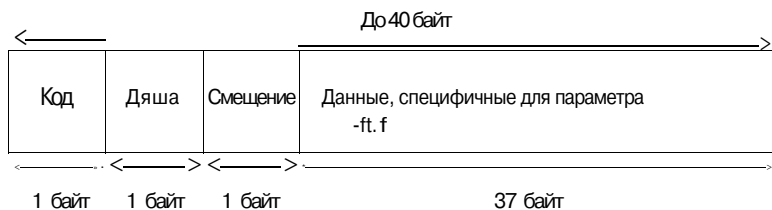
Параметры сокета на уровне *IPPROTOJP* относятся к специфичным атрибутам протокола IP, таким как изменение определенных полей в заголовке IP и добавление сокета к многоадресной группе IP. Многие из этих параметров объявлены в Winsock h и Winsock2 h по-разному. Заметьте, что при загрузке Winsock 1 необходимо включить правильный заголовок и компоновать приложение с Wsock32 lib. Аналогичным образом следует включить файл заголовка Winsock 2 и компоновать приложение с Ws2\_32 lib для Winsock 2. Обсуждаемые в этом разделе параметры часто используются для многоадресного вещания, поддерживаемого обеими версиями. Многоадресное вещание поддерживают все платформы Win32, кроме Windows CE до версии 2.1.

### Параметр *IP\_OPTIONS*

Этот параметр (тип *optval* — *char[]* версия Winsock 1+) можно и пол\чить и задать. Он позволяет задавать разные параметры IP внутри заголовка IP. Некоторые из возможных вариантов

- i **Ограничения обработки и защиты** — согласно RFC 1108,
- Запись маршрута** — каждый маршрутизатор добавляет свой IP адрес к заголовку (см. пример использования Ping в главе 13),
  - **штамп времени** — каждый маршрутизатор добавляет свой IP адрес и время,
- Нестрогая маршрутизация источника** — пакет требуется для посещения каждого из перечисленных в заголовке параметра IP-адресов,
  - **строгая маршрутизация источника** — пакет требуется для посещения только тех IP-адресов, которые перечислены в заголовке параметра

Учтите, что не все эти параметры поддерживаются узлами и маршрутизаторами. Когда вы определяете параметр IP, данные, передаваемые в вызов *setsockopt*, следуют за структурой (рис. 9-2). Допустимая длина заголовка параметра IP — до 40 байт.



**Рис. 9-2. Формат заголовка параметра IP**

Поле кода указывает тип параметра. Например, значение 0x7 представляет параметр <запись маршрута>. Длина — это просто длина заголовка параметра, а смещение — значение смещения в заголовке, где начинается фрагмент данных заголовка. Вид фрагмента данных заголовка зависит от конкретного параметра.

В приведенном далее отрывке кода мы задаем параметр <запись маршрута>. Сначала объявляем структуру (*struct ip\_option\_hdr*), содержащую первые три значения параметра (код, длина, смещение), а затем — данные параметра. Так как массив состоит из девяти длинных целых чисел без знака, можно записать до девяти IP-адресов. Помните, что максимальный размер заголовка параметра IP составляет 40 байт. Впрочем, наша структура занимает только 39 байт, поэтому система дополнит за вас заголовок до 32-разрядного слова (до 40 байт).

```
struct ip_option_hdr
{
    unsigned char    code,
    unsigned char    length,
    unsigned char    offset,
```

к

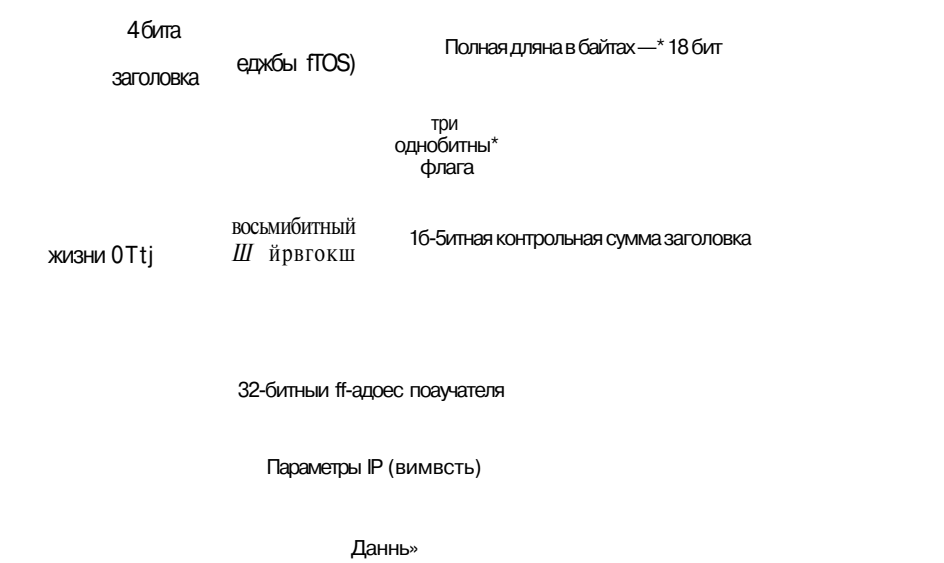
```
    unsigned long    addr[9],
} ophdr,

ZeroMemory((char *)&ophdr, sizeof(ophdr));
ophdr code = 0x7,
ophdr length = 39;
ophdr offset = 4, // Смещение на первый адрес (addr)
ret = setsockopt(s, IPPROTO_IP, IP_OPTIONS, (char *)&ophdr,
    sizeof(ophdr));
```

После того как параметр задан, он применяется к любым пакетам, отправляемым на данный сокет. Также для обнаружения заданных параметров вы можете вызвать *getsockopt* с *IP\_OPTIONS*, впрочем, при этом не будут возвращены данные, хранимые в специфичных для параметра буферах. Для извлечения данных, заданных в параметрах IP, необходимо использовать простые сокеты (*SOCK\_RAW*). Или задать параметр *IP\_HDRINCL* — тогда заголовок IP возвращается вместе с данными после вызова функции приема данных Winsock.

**Параметр IP\_HDRINCL**

Этот параметр (тип *optval* — *BOOL*, версия Winsock 2+) можно и получить, и задать. Если он равен *TRUE*, то функция отправки включит заголовок IP перед посылаемыми данными, а функция приема — вернет этот заголовок вместе с данными. Так что при вызове функции *send* Winsock включите полный заголовок IP перед данными и правильно заполните каждое поле заголовка (рис. 9-3). Этот параметр доступен только в Windows 2000.



**Рис. 9-3. Заголовок IP**

Первое поле заголовка — версия IP (на данный момент, версия 4) Длина заголовка — количество 32-битных слов в заголовке Заголовок IP всегда должен быть кратен 32 бшам

Следующее поле — тип службы, подробнее о котором — в описании параметра сокета *IP\_TOS* Поле полной длины выражает длину заголовка IP и данных в байтах, а затем идет уникальный идентификатор каждого отправленного IP-пакета Как правило, система увеличивает это значение с отправкой каждого пакета Поля флагов и смещения используются при разбивке IP-пакетов на пакеты меньшего размера Время жизни (поле TTL) ограничивает количество маршрутизаторов, через которые может пройти пакет При отправке пакета поле TTL каждый раз уменьшается маршрутизатором на 1, и когда значение достигает 0, передача прекращается Это ограничивает время, в течение которого пакет может передаваться по сети Поле протокола используется для демultipлексирования входящих пакетов Вот несколько допустимых протоколов, которые используют адресацию IP TCP, UDP, IGMP и ICMP

Далее идет 16-битная контрольная сумма заголовка Она рассчитывается только для заголовка, без учета данных Следующие два поля — 32-битные IP-адреса отправителя и получателя Поле параметров IP имеет переменную длину и содержит дополнительные сведения (обычно о защите или маршрутизации)

Самый простой способ отправить заголовок IP вместе с данными — оп-ределить структуру, содержащую заголовок IP и данные, и передать ее в вы-зов *send* Этот параметр работает только в Windows 2000 (подробнее — в главе 13)

## Параметр *IPJTOS*

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и получить, и задать Он представляет собой *тип службы* (type of service, TOS) — поле в заголовке IP, используемое для обозначения некоторых характеристик пакета Длина поля — 8 бит, оно разбито на три части трехбитное поле старшинства (которое игнорируется), четырехбитное поле TOS, и оставшийся бит (который должен быть равен 0) Назначение четырех бит TOS — сократить задержку, увеличить производительность, повысить надежность и сократить издержки, причем одновременно нельзя задать более одного бита Если значение всех четырех бит равно 0, то подразумевается обычное обслуживание RFC 1340 определяет биты, рекомендуемые для разных стандартных приложений типа TCP, SMTP, NNTP и т п В RFC 1349 исправлены некоторые положения RFC 1340

Для интерактивных приложений (например, Rlogin или Telnet) может потребоваться сократить задержку Для любого вида передачи файлов, например, FTP, важна максимальная производительность, а высокая надежность необходима для управления сетью (SNMP) и протоколов маршрутизации Наконец, новости Usenet (NNTP) — пример снижения издержек Параметр *IPJTOS* не доступен в Windows CE

При установке бит TOS на соquete, поддерживающем QoS, возникает дополнительная проблема. Поскольку QoS использует старшинство IP для различения уровней обслуживания, изменение этих значений разработчиками нежелательно. Поставщик услуг QoS перехватит вызов *setsockopt* с *IP\_TOS* на QoS-соquete, чтобы проверить возможность замены. Подробнее о QoS — в главе 12.

### Параметр *IP\_TTL*

Этот параметр (тип *optval* — *int*, версия Winsock 1+), представляющий собой IP-параметр времени жизни, можно и получить, и задать. *Поле времени жизни* (time-to-live, TTL) представлено в заголовке IP. Дейтаграмма использует поле TTL, чтобы ограничить количество маршрутизаторов, через которые она может пройти, дабы предотвратить циклы маршрутизации (когда дейтаграмма передается по кругу). Каждый маршрутизатор, через который проходит дейтаграмма, уменьшает ее поле TTL на 1. После обнуления TTL дейтаграмма отбрасывается. Этот параметр не доступен в Windows CE.

### Параметр *IP\_MULTICAST\_IF*

Этот параметр многоадресного интерфейса (тип *optval* — *unsigned long*, версия Winsock 1+) и получает, и задает локальный интерфейс, откуда локальный компьютер может отправлять любые данные для группы. Его имеет смысл использовать только на компьютерах с несколькими сетевыми интерфейсами (сетевыми адаптерами, модемами и т. п.). Параметр *optval* должен быть длинным целым без знака, представляющим двоичный IP-адрес локального интерфейса. Для преобразования строкового адреса в десятично-точечной нотации в длинное целое без знака используйте функцию *inet\_addr*.

```
DWORD    mcastIF;
```

```
// Сначала присоедините сокет к многоадресной группе
mcastIF = inet_addr("129.113.43.120");
ret = setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&mcastIF,
    sizeof(mcastIF));
```

### Параметр *IP\_MULTICAST\_TTL*

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и получить, и задать. Он определяет время жизни на пакетах многоадресной рассылки для этого сокета и напоминает уже знакомый вам TTL IP, но в отличие от последнего, применяется только к групповым данным, отправленным с данного сокета. TTL также предотвращает циклы маршрутизации, но при многоадресном вещании сужает область распространения данных. Поэтому для получения дейтаграмм члены многоадресной группы должны находиться в пределах «диапазона». Стандартное значение TTL для групповых дейтаграмм — 1.

### Параметр `IP_MULTICAST_LOOP`

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE* (по умолчанию), то данные, отправленные на групповой адрес, будут отражены во входящем буфере сокета. Если присвоить этому параметру значение *FALSE*, то никакие отправленные сокетом данные не попадут в его очередь входящих данных.

### Параметр `IP_ADD_MEMBERSHIP`

Этот параметр (тип *optval* — *struct ipjnreq*, версия Winsock 1+) можно только задать — это способ добавить сокет к многоадресной группе IP в Winsock 1. Создайте сокет с семейством адресов *AFJNET* и типом *SOCKJDGRAM* в вызове функции *socket*. Для добавления сокета в многоадресную группу, используйте структуру:

```
struct ipjnreq
{
    struct in_addr  imrjnultiaddr;
    struct m_addr  imr_interface;
};
```

Здесь поле *imrjnultiaddr* — двоичный групповой адрес, а *imr\_interface* — локальный интерфейс, на котором следует отправлять и получать групповые данные. Подробнее о допустимых групповых адресах — в главе 11. Поле *imrjnterface* — двоичный IP-адрес локального интерфейса, либо значение *INADDR\_ANY* для выбора интерфейса по умолчанию.

### Параметр `IP_DROP_MEMBERSHIP`

Этот параметр (тип *optval* — *struct ipjnreq*, версия Winsock 1+) можно только задать. Он противоположен по назначению *IP\_ADD\_MEMBERSHIP* и удаляет сокет из данной группы IP. При его вызове со структурой *ipjnreq*, содержащей значения, которые использовались для регистрации в данной группе, сокет 5 будет удален из нее. Подробнее — в главе 11.

### Параметр `IP_DONTFRAGMENT`

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, сети запрещается фрагментировать дейтаграмму IP в ходе передачи. Впрочем, если размер дейтаграммы больше *максимального блока передачи данных* (maximum transmission unit, MTU) и флаг IP «не фрагментировать» задан внутри заголовка IP, отправка будет приостановлена и появится сообщение об ошибке ICMP и необходимости фрагментации. Этот параметр не доступен в Windows CE.

## Уровень параметров *IPPROTO\_TCP*

К этому уровню относится лишь один параметр, применимый только к потоковым сокетам (*SOCK\_STREAM*), принадлежащим к семейству адресов *AFJNET*. Он доступен во всех версиях Winsock и поддерживается всеми платформами Win32.



### Параметр **TCP\_NODELAY**

Чтобы увеличить эффективность и пропускную способность за счет уменьшения издержек, система выполняет алгоритм Nagle, который позволяет «укрупнить» пакеты TCP. Издержки в том, что длина заголовка TCP для каждого пакета — 20 байт. Расточительно отправлять сообщения по 2 байта с 20-байтовым заголовком, поэтому когда приложение запрашивает порцию данных, система сможет накапливать данные в течение некоторого времени перед фактической отправкой по сети. Если дополнительные данные за указанный период времени не накопятся, отправка произойдет безусловно.

Недостаток этого алгоритма — замедление подтверждений о получении данных TCP. Но поскольку узел будет ждать накопления данных для отправки партнеру, он может присоединить уведомление ACK к следующей порции данных, а не отправить его сразу в отдельном пакете.

Параметр **TCP\_NODELAY** (тип *optval* — **BOOL**, версия Winsock 1+) можно и получить, и задать. Если он равен **TRUE**, алгоритм Nagle отключается. Этот алгоритм порой отрицательно воздействует на какое-либо сетевое приложение, которое отправляет данные в относительно небольшом количестве и ожидает своевременного ответа. Классический пример — Telnet, интерактивное приложение, позволяющее пользователю войти на удаленную машину и давать ей команды. Как правило, пользователь нажимает на клавиши несколько раз в секунду, и алгоритм Nagle такой сеанс просто бы «тормозил».

### Уровень параметров **NSPROTO\_IPX**

Описанные в этом разделе параметры сокета специфичны для Microsoft-расширения интерфейса сокетов Windows IPX/SPX и предназначены для совместимости с существующими приложениями. Впрочем, их не рекомендуют использовать, поскольку они гарантированно работают только со стеком Microsoft IPX/SPX. Приложение, использующее эти расширения, не сможет работать в других реализациях IPX/SPX. Эти параметры определены в файле **WSNwLink.h**, который должен быть включен после **Winsock.h** и **Wsipx.h**.

#### Параметр **IPX\_PTYPE**

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и получить, и задать. Значение аргумента *optval* будет определять тип каждого пакета IPX, отправленного с этого сокета.

#### Параметр **IPX\_FILTERPTYPE**

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно и получить, и задать. Он определяет тип пакета для фильтрации получаемых данных. При любом вызове приема возвращаются только пакеты IPX с типом, указанным в *optval*, остальные — отбрасываются.

#### Параметр **IPX\_JSTOPFILTERPTYPE**

Этот параметр (тип *optval* — *int*, версия Winsock 1+) можно только задать. Он позволяет прекратить фильтрацию типов пакетов, заданную параметром **IPX\_FILTERPTYPE**.

**Параметр *IPX\_DSTYPE***

Этот параметр (тип *optval* — *mt*, версия Winsock 1+) можно и получить, и задать. Он представляет значение поля потока данных в заголовке SPX каждого отправленного пакета.

**Параметр *IPX\_EXTENDED\_ADDRESS***

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Он разрешает (если равен *TRUE*) или запрещает расширенную адресацию пакетов IPX. При отправке он добавляет элемент *unsigned char saJitype* к структуре *SOCKADDRJIPX*, в результате чего ее полная длина достигает 15 байт. При приеме к структуре *SOCKADDRJIPX* сразу добавляются и *saJitype*, и *unsigned char saJlags*, после чего длина ее достигает 16 байт. Текущие биты, определенные в *saJlags*:

- III 0x01* — полученный кадр был отправлен путем широковещания,
- *0x02* — полученный кадр была отправлен с этого компьютера

**Параметр *IPXJRECVHDR***

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если он равен *TRUE*, то вызов любой функции приема Winsock возвращает вместе с данными заголовок IPX.

**Параметр *IPXJVIAXSIZE***

Этот параметр (тип *optval* — *mt*, версия Winsock 1+) **можно только получить**. Вызов *getsockopt* возвращает максимально возможный размер дейтаграммы IPX.

**Параметр *IPX\_ADDRESS***

Этот параметр (тип *optval* — *LPX\_ADDRESSJDATA*, версия Winsock 1+) можно только получить. Он запрашивает информацию об определенном адаптере, связанном с IPX. В системе с *n* адаптерами они пронумерованы от 0 до *n-1*. Чтобы узнать количество IPX-адаптеров в системе, используйте параметр *IPX\_MAX\_ADAPTERJNUM* с функцией *getsockopt* или вызывайте *IPX\_ADDRESS* с увеличением значений *adaptemum*, пока не будет возвращена ошибка. Параметр *optval* указывает на структуру *IPX\_ADDRESSJDATA*.

```
typedef struct _IPX_ADDRESS DATA
```

```
{
    INT    adapternum    // Ввод  Номер адаптера, начинающийся с 0
    UCHAR  netnum[4],    // Вывод  Номер сети IPX
    UCHAR  nodenum[6]    // Вывод  IPX-адрес узла
    BOOLEAN wan,         // Вывод  TRUE = адаптер на ГВС-канале
    BOOLEAN status,      // Вывод  TRUE = связь по ГВС отсутствует (или адаптер
                        // не связан с ГВС)
    INT    maxpkt;       // Вывод  максимальный размер пакета, исключая заголовок
IPX
    t/LONG  linkspeed;   // Вывод  скорость связи в 100 байт/сек
}
```

```

// (т.е. 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDORESS_DATA;

```

### Параметр **IPX\_GETNETINFO**

Этот параметр (тип *optval* — *IPX\_NETNUM\_DATA*, версия Winsock 1+) можно только получить. Он возвращает информацию относительно определенного номера сети IPX. Если сеть — в кэше IPX, параметр возвращает информацию напрямую. В ином случае рассылаются RIP-запросы, чтобы ее обнаружить. Параметр *optval* указывает на действительную структуру *IPX\_NETNUM\_DATA*.

```

typedef struct _IPX_NETNUM_DATA
{
    UCHAR netnum[4]; // Ввод. Номер сети IPX
    USHORT hopcount; // Вывод. Количество транзитов для этой сети по
                    // порядку компьютеров
    USHORT netdelay; // Вывод. количество "тиков" для этой сети по
                    // порядку компьютеров
    INT cardnum; // Вывод: начинающийся с 0 номер адаптера, используемый
                // для
                    // направления в эту сеть; может быть использован как
                    // значение для adapternum в поле IPX_ADDRESS
    UCHAR router[6]; // Вывод. MAC-адрес маршрутизатора следующего транзита,
                    // равен 0, если сеть уже достигнута
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;

```

### Параметр **IPX\_GETNETINFO\_NORIP**

Этот параметр (тип *optval* — *IPX\_NETNUM\_DATA*, версия Winsock 1+) можно и получить, и задать. Если возвращается *TRUE*, то дейтаграммы не фрагментируются IP. Этим он напоминает *IPX\_GETNETINFO*, но *IPX\_NETNUM\_DATA* не рассылает RIP-запросы. Если сеть находится в кэше IPX, то параметр возвращает информацию, иначе выдается ошибка (см. также параметр *IPX\_RERIP-NETNUMBER*, который всегда рассылает RIP-запросы). Как и в случае *IPX\_GETNETINFO*, для использования этого параметра требуется, чтобы параметр *optval* указывал на структуру *IPX\_NETNUM\_DATA*.

### Параметр **IPX\_SPXGETCONNECTIONSTATUS**

Этот параметр (тип *optval* — *IPX\_SPXCONNSTATUS\_DATA*, версия Winsock 1+) можно только получить. Он возвращает информацию о связанном SPX-сокетe. Параметр *optval* указывает на структуру *IPX\_SPXCONNSTATUS\_DATA*. Все числа даны в сетевом порядке байт (от высокого к низкому).

```

typedef struct _IPX_SPXCONNSTATUS_DATA
{
    UCHAR ConnectionState;
    UCHAR WatchDogActive;
    USHORT LocalConnectionId;
    USHORT RemoteConnectionId;
    USHORT LocalSequenceNumber;
}

```

```

USHORT LocalAckNumber;
USHORT LocalAllocNumber;
USHORT RemoteAckNumber;
USHORT RemoteAllocNumber;
USHORT LocalSocket;
π'  UCHAR ImmediateAddress[6];
    UCHAR RemoteNetwork[4];
ο', UCHAR RemoteNode[6];
    USHORT RemoteSocket;
    USHORT RetransmissionCount;
    USHORT EstimatedRoundTripDelay; /* In milliseconds */
    USHORT RetransmittedPackets;
    USHORT SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;

```

### Параметр *IPX,,ADDRESS^NOTIFY*

Этот параметр (тип *optval* — *IPX\_ADDRESS\_DATA*, версия Winsock 1+) можно только получить. Он асинхронно уведомляет об изменении состояния связанного с IPX адаптера, что обычно происходит, когда ГВС-канал включается или разрывается. Для использования параметра требуется, чтобы вызывающая программа предоставила в параметре *optval* структуру *IPX\_ADDRESS\_DATA*. Особенность в том, что сразу за структурой *IPX\_ADDRESS\_DATA* следует описатель занятого события. Вот один из вариантов вызова этого параметра-  
char buff[sizeof(IPX\_ADDRESS\_DATA) + sizeof(HANDLE)];

IPX\_ADDRESS\_DATA <ipxdata;

HANDLE \*hEvent;

```

ipxdata = (IPX_ADDRESS_DATA *)buff;
hEvent = (HANDLE *) (buff + sizeof(IPX_ADDRESS_DATA));
ipxdata->adapternum = 0; // укажите соответствующий адаптер
•hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
setsockopt(s, NSPROTO_IPX, IPX_ADDRESS_NOTIFY, (char *)buff,
sizeof(buff));

```

После успешного опроса *getsockopt* структура *IPX ADDRESS DATA*, на которую указывает *optval*, не изменится. Вместо этого запрос помещается во внутреннюю очередь транспорта, а при изменении состояния адаптера IPX находит его и заполняет все поля в структуре *IPX ADDRESS DATA*. Затем событие, на которое ссылается описатель в буфере *optval*, помечается как свободное. Если одновременно выполняется несколько вызовов *getsockopt*, используйте разные события. Событие применяется, поскольку запрос должен быть асинхронным, что на данный момент *getsockopt* не поддерживает.

**ВНИМАНИЕ!** В текущей реализации транспорт сообщает только об одном поставленном в очередь запросе для каждого изменения состояния. Поэтому не запускайте больше одной службы, ставящей запросы в очередь.

### Параметр *IPXJAAX\_ADAPTER\_NUM*

Этот параметр (тип *optval* — *mt*, версия Winsock 1+) можно только получить. Он возвращает количество адаптеров IPX. Если при вызове возвращается *n* адаптеров, они нумеруются от 0 до *n*—1.

### Параметр *IPX\_RERIPNETNUMBER*

Этот параметр (тип *optval* — *IPX\_NETNUM\_DATA*, версия Winsock 1+) можно только получить. Он возвращает информацию о номере сети. Напоминает *IPX\_GETNETINFO*, но вынуждает IPX переиздавать RIP-запросы, даже если сеть находится в кэше (но не в случае, когда сеть присоединена напрямую). Как и *IPX\_GETNETINFO*, *IPX\_NETNUM\_DATA* требует, чтобы *optval* указывал на структуру *IPX\_NETNUM\_DATA*.

### Параметр *IPX\_RECEIVE\_BROADCAST*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно только задать. По умолчанию IPX-сокет может принимать широковещательные пакеты. Если этот параметр равен *TRUE*, широковещательные IPX-пакеты приниматься не будут. Приложениям, которые не должны получать эти пакеты, следует присвоить *IPX\_RECEIVE\_BROADCAST* значение *FALSE*, чтобы увеличить быстродействие системы. Впрочем, учтите, что это не обязательно вызовет фильтрацию широковещания для приложения.

### Параметр *IPX\_IMMEDIATESPXACK*

Этот параметр (тип *optval* — *BOOL*, версия Winsock 1+) можно и получить, и задать. Если присвоить ему *TRUE*, пакеты подтверждения для соединений SPX не будут задерживаться. Это увеличивает количество ACK, но визуально ускоряет работу ряда программ.

## Функции *ioctlsocket* и *WSAIoctl*

Сокетные *ioctl*-функции используются для управления режимом ввода-вывода, а также для получения сведений об ожидающем вводе-выводе. Первая функция — *ioctlsocket*, появилась в спецификации Winsock 1.

```
int ioctlsocket (
    SOCKET s,
    long cmd,
    u_long FAR *argp
),
```

Параметр *s* — описатель сокета, который будет использоваться в дальнейшем, а *cmd* — предопределенный флаг для команды управления вводом-выводом, которая будет выполняться. Последний параметр — *argp*, указатель на переменную, специфичную для данной команды. После описания всех команд дается тип требуемой переменной.

В Winsock 2 появилась функция *ioctl*, которая добавляет несколько новых параметров. Во-первых, она разбивает один параметр *argp* на набор вход-

ных параметров для значений, переданных в функцию, и набор выходных параметров, используемых для возвращения данных из вызова. Также вызов этой функции может использовать перекрытый ввод-вывод. Функция называется *WSAloctl* и определена так:

```
int WSAloctlK
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    |   DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    *   LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
```

; > \*

Первые два параметра совпадают с параметрами *ioctlsocket*. Следующие два — *lpvInBuffer* и *cbInBuffer*, описывают входные параметры. Параметр *lpvInBuffer* — это указатель на передаваемое значение, а *cbInBuffer* — размер этих данных в байтах. Аналогично параметры *lpvOutBuffer* и *cbOutBuffer* используются для возвращения данных из вызова. Параметр *lpvOutBuffer* указывает на буфер данных, куда помещается любая возвращенная информация. Параметр *cbOutBuffer* задает размер буфера из поля *lpvOutBuffer* в байтах. Заметьте, некоторые вызовы могут использовать только параметры ввода или вывода, а некоторые — и те, и другие. Седьмой параметр — *lpcbBytesReturned*, возвращает количество фактически возвращенных байт. Последние два параметра — *lpOverlapped* и *lpCompletionRoutine*, используются при вызове этой функции с перекрытым вводом-выводом. Подробнее о перекрытом вводе-выводе — в главе 8.

## Стандартные ioctl-команды

Эти три ioctl-команды <перекочевали> из мира Unix и употребляются чаще всего. Они доступны на всех платформах Win32 и могут быть вызваны с использованием как *ioctlsocket*, так и *WSAloctl*.

### Команда FIONBIO

Используется с функциями *ioctlsocket* и *WSAloctl* (версия Winsock 1+). Тип входного значения — *unsigned*. Эта команда включает или отключает неблокирующий режим на сокете. По умолчанию все сокеты с момента создания — блокирующие. Для включения неблокирующего режима при вызове *ioctlsocket* с ioctl-командой *FIONBIO* задайте *argp* для передачи указателя на длинное целое без знака с ненулевым значением. Значение 0 переводит сокет в блокирующий режим. Если вместо этого вы используете *WSAloctl*, просто передайте в качестве параметра *lpvInBuffer* длинное целое без знака.

Вызов функции *WSAAsyncSelect* или *WSAEventSelect* автоматически разблокирует сокет, и любая попытка перевести его обратно в блокирующий режим вернет ошибку *WSAEINVAL*. Чтобы вернуть сокет в блокирующий режим,

приложение должно сначала отключить *WSAAsyncSelect*, вызвав *WSAAsyncSelect* с параметром *lEvent* равным 0 Другой способ — отключить *WSAEventSelect*, вызвав *WSAEventSelect* с параметром *INetworkEvents*, равным 0

### Команда **FIONREAD**

Обе функции — *ioctlsocket* и *WSAIoctl* (версия Winsock 1+), возвращают значение типа *unsigned long*, на входе ничего указывать не требуется Команда *FIONREAD* определяет количество данных, которые могут быть автоматически считаны с сокета Для *ioctlsocket* значение *argp* выдается с длинным целым без знака и определяет количество байт, которое должно быть прочитано При использовании *WSAIoctl* целое без знака возвращается в *ipvOutBuffer* Если сокет *s* — поточный (*SOCK\_STREAM*), *FIONREAD* возвращает конечное количество данных, которое можно прочитать в одном вызове функции приема Запомните использование этого или других механизмов приема сообщений не всегда гарантирует выдачу точного количества байт Когда эта *ioctl*-команда используется на дейтаграммном сокете (*SOCK\_DGRAM*), возвращаемое значение — размер первого сообщения, стоящего в очереди на соке

### Команда **SIOCATMARK**

Обе функции (версия Winsock 1+) возвращают значение типа *BOOL*, на входе ничего указывать не требуется Команда *SIOCATMARK* определяет, были ли прочитаны *внешние* (out-of-band, OOB) данные После настройки сокета для приема OOB-данных вместе с другими (с помощью параметра сокета *SO\_OOBINLINE*) эта *ioctl*-команда возвращает *TRUE*, если следующими должны быть прочитаны OOB-данные Иначе возвращается *FALSE* и следующая операция приема вернет все или некоторые данные, которые предшествуют OOB-данным Для *ioctlsocket* указатель на булеву переменную возвращается в *argp*, а для *WSAIoctl* — в *IpvOutBuffer* Помните, что в одном вызове функции приема OOB-данные никогда не смешиваются с остальными Подробнее об OOB-данных — в главе 7

## Другие *ioctl*-команды

Эти *ioctl*-команды специфичны для Winsock 2 кроме тех, которые работают с SSL и доступны только в Windows CE Если вы просмотрите заголовки Winsock 2, то фактически увидите объявление и других *ioctl*-команд Впрочем, только *ioctl*-команды, перечисленные в этом разделе, значимы для пользователейских приложений К тому же не все *ioctl*-команды работают на всех платформах Win32, хотя, конечно, это может измениться по мере обновлений ОС Для Winsock 2 большинство этих команд определено в Winsock2.h Некоторые из новейших *ioctl*-команд, специфичных для Windows 2000, определены в Mstcpip.h

### Команда **SIO\_ENABLE\_CIRCULAR\_QUEUEING**

Функция *WSAIoctl* (версия Winsock 2+) принимает и возвращает значение типа *BOOL* Эта *ioctl*-команда контролирует, как базовый поставщик службы

обрабатывает входящие дейтаграммные сообщения, если очередь заполнена По умолчанию, в этом случае любые входящие дейтаграммы отбрасываются Когда этот параметр равен *TRUE*, вновь прибывшие сообщения не будут пропущены из-за переполнения буфера старые сообщения в очереди удаляются, чтобы освободить место для новых Эта команда допустима только для сокетов, связанных с ненадежными, ориентированными на сообщения протоколами Если она используется на сокете другого типа (например, поточном) или поставщик службы ее не поддерживает, выдается ошибка *WSAENOPROTOOPT* Этот параметр поддерживает только Windows NT/2000

Команда *SIO\_ENABLE\_CIRCULAR\_QUEUEING* также может быть использована для включения (выключения) циклической очереди или с целью опроса текущего состояния параметра Чтобы задать значение, нужно использовать только входные параметры При опросе текущего значения параметра требуется только выходной параметр типа *BOOL*

### Команда *SIO\_FIND\_ROUTE*

Функция *WSAIoctl* (версия Winsock 2+) принимает значение типа *SOCKADDR* и возвращает значение типа *BOOL* Эта ioctl-команда используется для проверки, можно ли связаться с определенным адресом по сети Параметр *IpvInBuffer* указывает на структуру *SOCKADDR* данного протокола Если адрес уже существует в локальном кэше, он становится недействительным Для IPX вызов функции *WSAIoctl* инициирует вызов *IPX\_GetLocalTarget*, который запрашивает в сети этот удаленный адрес К сожалению, поставщик Microsoft для текущих платформ Win32 не реализует эту команду

### Команда *SIO\_FLUSH*

Функция *WSAIoctl* (версия Winsock 2+) не требует входных или выходных параметров Эта ioctl-команда определяет, были ли прочитаны ООБ-данные и отбрасывает текущее содержимое очереди отправки, связанной с данным сокетом В настоящее время этот параметр реализован только в Windows 2000 и NT 4 Service Pack 4

### Команда *SIO\_GET\_BROADCAST\_ADDRESS*

Функции *WSAIoctl* (версия Winsock 2+) входные параметры не требуются Возвращается структура *SOCKADDR* (через *ipvOutBuffer*), содержащая широковещательный адрес для семейства адресов из сокета *s*, который может быть использован в *sendto* или *WSASendTo* Работает только в Windows NT/2000 Windows 9x возвращает ошибку *WSAEINVAL*

### Команда *SIO\_GET\_EXTENSION\_FUNCTION\_POINTER*

Функция *WSAIoctl* (версия Winsock 2+) принимает значение типа *GUID* и возвращает указатель на функцию Эта ioctl-команда используется для доступа к функциям, специфичным для поставщика, но не входящим в спецификацию Winsock Через нее поставщик может предоставить программистам доступ к функциям, назначив каждой из них GUID Затем приложение получит указатель на нужную функцию с помощью ioctl-команды *SIO\_GET\_EX-*



*TENSION\_FUNCTION\_POINTER* Заголовочный файл *Mswsock.h* определяет добавленные Microsoft функции Winsock, включая их GUID. Например, чтобы выяснить, поддерживает ли установленный поставщик Winsock функцию *TransmitFile*, опросите его с использованием GUID

```
«define WSAID.TRANSMITFILE \
    {0xb5367df0,0xcbac,0x11cf,{0x95,0xca,0x00,0x80,0x5f,0x48,0xa1,0x92}}
```

Получив указатель для функции расширения, например *TransmitFile*, вы сможете вызывать ее напрямую, без компоновки приложения с библиотекой *Mswsock.lib*. Это фактически исключит один промежуточный вызов функции, выполняемый в *Mswsock.lib*.

Вы можете найти в *Mswsock.h* и другие специфичные для Microsoft расширения с определенными для них GUID. Эта *ioctl*-команда — важная часть разработки многоуровневого поставщика службы. Подробнее об интерфейсе поставщика службы — в главе 14.

### Команда *SIOJCHKJQOS*

Функция *WSAIoctl* (версия Winsock 2+) принимает и возвращает значение типа *DWORD*. Эта *ioctl*-команда задает атрибуты QoS для данного сокета и может применяться для проверки шести состояний QoS. Пока она поддерживается только в Windows 2000. Шести состояниям соответствуют флаги *ALLOWED\_TO\_SEND\_DATA*, *ABLE\_TO\_RECV\_RSV*, *LINE\_RATE*, *LOCAL\_TRAFFIC\_CONTROL*, *LOCAL\_QOSABILITY* и *ENDTO\_END\_QOSABILITY*.

Первый флаг — *ALLOWED\_TO\_SEND\_DATA*, используется после задания уровней QoS на сокет с помощью *SIO\_SET\_QOS*, но до получения запроса резервирования (RESV) протокола RSVP. Перед этим потоку, соответствующему сокету, обеспечивается лишь *негарантированное* (best-effort) обслуживание. Получение сообщения RESV показывает, что требования к пропускной способности удовлетворены. Подробнее о протоколе RSVP и резервировании сетевых ресурсов — в главе 12.

Флаг *ALLOWED\_TO\_SEND\_DATA* позволяет увидеть, достаточно ли негарантированного обслуживания для уровней QoS, запрошенных *SIO\_SET\_QOS*. Возвращенное значение будет либо 1 — текущая негарантированная пропускная способность достаточна, либо 0 — пропускная способность не обеспечивает требуемые уровни. Если флаг возвращает 0, приложение перед отправкой данных должно подождать, пока не будет получено сообщение RESV.

Второй флаг — *ABLE\_TO\_RECV\_RSV*, показывает, способен ли узел принимать и обрабатывать RSVP-сообщения на интерфейсе, с которым связан данный сокет. Возвращаемое значение информирует, могут (1) или не могут (0) приниматься RSVP-сообщения.

Следующий флаг — *LINE\_RATE*, возвращает негарантированную скорость передачи в Кбит/с. Если скорость неизвестна, возвращается значение *INFO\_NOT\_AVAILABLE*.

Последние три флага показывают, существуют ли определенные возможности на локальном компьютере или в сети. Все три параметра выдают 1 — если параметр поддерживается, 0 — если нет, *INFO\_NOT\_AVAILABLE* — если

не существует способа проверки *LOCAL TRAFFIC CONTROL* используют, чтобы определить, установлен и доступен ли на компьютере компонент Traffic Control (управление трафиком) *LOCAL\_QOSABILITY* определяет, поддерживается ли QoS на локальном компьютере. И наконец, *END\_TO\_END\_QOSABILITY* показывает, допустимо ли применение QoS в локальной сети

### Команда ***SIO\_GET\_QOS***

Функция *WSAIoctl* (версия Winsock 2+) возвращает значения типа *QOS*. Эта *ioctl*-команда получает структуру *QOS*, связанную с сокетом. Предоставленный буфер должен быть достаточно объемным, чтобы вместить в себя эту структуру целиком. Между тем ее размеры в ряде случаев превышают *sizeof(QOS)*, так как могут содержать информацию, специфичную для поставщика. Подробнее о QoS — в главе 12. Если эта *ioctl*-команда используется на соquete, семейство адресов которого не поддерживает QoS, выдается ошибка *WSAENOPROTOOPT*. Этот параметр и *SIO\_SET\_QOS* доступны только на платформах, обеспечивающих QoS-совместимый транспорт, таких как Windows 98 и Windows 2000.

### Команда ***SIOJSETJ3OS***

Эта *ioctl*-команда составляет пару для *SIO\_GET\_QOS*. Входной параметр для функции *WSAIoctl* (версия Winsock 2+) — структура *QOS*, которая определяет требования к пропускной способности для данного сокета. Выходные значения не возвращаются. Этот параметр и *SIO\_GET\_QOS* доступны только на платформах, обеспечивающих QoS-совместимый транспорт, таких как Windows 98 и Windows 2000.

### Команда ***SIO,,MULTIPOINTJ.OOPBACK***

Функция *WSAIoctl* (версия Winsock 2+) принимает и возвращает значение типа *BOOL* и определяет, будут ли многоадресные данные возвращены на сокет. При отправке группе данные по умолчанию поступают в очередь входящих сообщений исходного сокета. Естественно, такая петля образуется, только если сокет также входит в целевую многоадресную группу. В настоящее время это поведение характерно только для IP, но не для ATM. Для отключения петли передайте *FALSE* во входном параметре *ipvInBuffer*. Чтобы получить текущее значение этого параметра, оставьте входное значение равным *NULL*, а в качестве выходного параметра укажите булеву переменную.

### Команда ***SIO\_MULTICAST\_SCOPE***

Функция *WSAIoctl* (версия Winsock 2+) принимает и возвращает значение типа *mt*. Эта *ioctl*-команда контролирует время жизни (или область распространения) многоадресных данных. Область распространения — это количество маршрутизируемых сегментов сети, которые разрешено пересечь данным. Значение по умолчанию — 1. Когда многоадресный пакет попадает в Маршрутизатор, значение TTL уменьшается на 1. Когда TTL достигает 0, пакет отбрасывается. Чтобы задать это значение, передайте целое с желае-

мым TTL как *ipvInBuffer*. Если же вы хотите просто получить текущее значение TTL, вызовите *WSAIoctl* с *ipvOutBuffer*, указывающим на целое число.

### Команда *SIO\_KEEPAIVE\_VALS*

Функция *WSAIoctl* (версия Winsock 2+) принимает и возвращает значения типа *tcp\_keepalive*. Эта *ioctl*-команда позволяет включить передачу сообщений о *сохранении соединения* (*keepalive*) для TCP для каждого соединения и задать интервал между их отправкой. Параметр сокета *SO\_KEEPAUVE* также включает передачу сообщений о сохранении соединения TCP, но интервал между ними задается в системном реестре, модификация которого затронет все процессы на компьютере. Команда *SIO\_KEEPAUVE\_VALS* позволяет задать интервал отдельно для каждого сокета. Для этого следует инициализировать на данном сокете структуру *tcp\_keepalive* и предать ее как буфер ввода:

```
struct tcp_keepalive
{
    u_long    onoff;
    u_long    keepalivetime;
    u_long    keepaliveinterval;
}
```

Значение полей структуры *keepalivetime* и *keepaliveinterval* идентичны значениям реестра, которые мы уже обсуждали в разделе, посвященном параметру *SO\_KEEPAIVE*. Для запроса текущих значений вызовите *WSAIoctl* с командой *SIO\_KEEPAIVE\_VALS* и предоставьте структуру *tcp\_keepalive* в качестве буфера вывода. Эта команда доступна только в Windows 2000.

### Команда *SIO\_RCVALL*

Функция *WSAIoctl* (версия Winsock 2+) принимает значение типа *unsigned int*, выходных параметров нет. Использование этой команды со значением *TRUE* позволяет данному сокету получать все IP-пакеты в сети. Для этого необходимо передать описатель сокета в *WSAIoctl*. Нужно, чтобы сокет был из семейства адресов *AF\_INET*, тип — *SOCK\_RAW*, протокол — *IPPROTO\_IP*. Сокет должен быть связан с явным локальным интерфейсом, то есть нельзя создать привязку *INADDR\_ANY*.

После привязки сокета и задания *ioctl*-команды вызовы *recv/WSARecv* возвращают IP-дейтаграммы. Помните, что для дейтаграмм следует предусмотреть достаточно вместительный буфер. Поскольку поле суммарного размера заголовка IP — 16-битная величина, максимальный теоретический предел — 65 535 байт, однако на практике *максимальная единица передачи* (*maximum transmission unit*, MTU) в сетях гораздо меньше.

Использование *SIO\_RCVALL* требует привилегий администратора на локальном компьютере. К тому же эта *ioctl*-команда действует только на компьютере с Windows 2000. Образец приложения на прилагаемом диске — *Rcvall.c*, иллюстрирует использование этой и двух других *ioctl*-команд *SIO\_RCVALL*.

### Команда *SIO\_RCVALL\_MCAST*

Эта команда похожа на *SIORCVALL* и также получает все многоадресные пакеты в сети. Функция *WSAIoctl* (версия Winsock 2+) принимает значение типа *unsigned int*, выходных параметров нет. Применяются те же правила использования, что и для *SIO\_RCVALL*: сокет, переданный в *WSAIoctl*, также должен быть создан под протокол *IPPROTOJGMP*. Единственное отличие.- возвращается только многоадресный IP-трафик, а не все IP-пакеты. Это означает, что возвращаются только IP-пакеты с адресов 224.0.0.0 — 239-255.255.255. Эта *ioctl*-команда доступна только в Windows 2000.

### Команда *SIO\_JtCVALLJGMPMCAST*

Эта *ioctl*-команда тоже напоминает *SIOJICVALL*, включая то, что сокет, переданный в *WSAIoctl*, должен быть создан под протокол *IPPROTOJGMP*. Функция *WSAIoctl* (версия Winsock 2+) принимает значение типа *unsigned int*, выходных параметров нет. Но по этой команде возвращаются только все IGMP-пакеты. Инструкции по использованию этого параметра вы найдете в разделе по *SIO\_RCVALL*. Эта команда доступна только в Windows 2000.

### Команда

Обе функции (версия Winsock 2+) принимают значения типа *SOCKADDR* (выходных параметров нет) и определяют, были ли считаны OOB-данные. Эта *ioctl*-команда позволяет найти адрес локального интерфейса, который следует использовать при отправке данных на удаленный компьютер. Адрес удаленной машины предоставляют в виде структуры *SOCKADDR*, как параметр *ipInBuffer*. Буфер *ipOutBuffer* должен быть достаточно большим, чтобы вместить массив из одной или более структур *SOCKADDR*, описывающих доступные локальные интерфейсы. Эту команду можно использовать для одноадресных или многоадресных конечных точек, а возвращенный интерфейс — в последующих вызовах *bind*.

В основе команды *SIO\_ROUTING\_INTERFACE\_QUERY* — plug-and-play возможности Windows 2000. Пользователь может вставлять (вынимать) сетевой адаптер формата PCMCIA, что повлияет на доступные приложению интерфейсы. Учтите это в приложении для Windows 2000.

Приложения не могут полагаться на постоянство информации, возвращенной *SIO\_ROUTING\_INTERFACE\_QUERY*. Так что используйте *ioctl*-команду *SIO\_ROUTING\_INTERFACE\_CHANGE*, которая уведомит приложение о смене интерфейсов — тогда еще раз вызовите *SIO\_ROUTING\_INTERFACE\_QUERY*, чтобы освежить информацию.

### Команда *SIO\_ROUTING\_INTERFACE\_CHANGE*

Эта *ioctl*-команда отправляет уведомление, когда изменился интерфейс для конечной точки, и таким образом помогает быть в курсе изменений на локальном интерфейсе маршрутизации, используемом для доступа к указанному удаленному адресу. При применении этой команды структура *SOCKADDR* Для удаленного адреса передается в буфер ввода, и после успешного завер-

шения никакие данные не возвращаются. Впрочем, если по каким-то причинам, интерфейс для этого маршрута изменится, то приложение будет оповещено и сможет вызвать *SIO\_ROUTINGINTERFACE\_QUERY*, чтобы определить, какой интерфейс использовать дальше.

Существует несколько способов вызова этой команды. Если сокет — блокирующий, то вызов *WSAIocctl* не завершится до момента смены интерфейса. В ином случае вернется ошибка *WSAEWOULDBLOCK*. Затем приложение ожидает событий изменения маршрута, вызывая *WSAEventSelect* или *WSAAsyncSelect* с флагом *FD\_ROUTINGINTERFACE\_CHANGE*, заданным в битовой маске сетевого события. Перекрытый ввод-вывод тоже можно использовать: предоставьте описатель события в структуре *WSAOVERLAPPED*, которая освободится после смены маршрутизации.

Адрес, указанной в структуре ввода *SOCKADDR*, задают явно или используя шаблон *INADDR\_ANY* — тогда вы узнаете о любых изменениях маршрутизации. Поскольку информация о маршрутизации довольно статична, в поставщиках есть параметр для игнорирования информации, предоставляемой приложениями в буфер ввода. В этом случае поставщики просто отсылают уведомление после каждого изменения интерфейса. В итоге лучше зарегистрироваться для получения уведомлений о любом изменении и просто вызывать *SIO\_ROUTING\_INTERFACE\_QUERY*, чтобы проверить, влияют ли эти изменения на приложение.

### Команда *SIO\_ADDRESS\_UST\_QUERY*

Эта *ioctl*-команда используется для получения списка адресов локального транспорта, соответствующих семейству протоколов сокета, с которым может связаться приложение. Функция *WSAIocctl* (версия Winsock 2+) не имеет значений ввода. Буфер вывода — структура *SOCKET\_ADDRESS\_LIST*:

```
typedef struct _SOCKET_ADDRESS_LIST

INT                iAddressCount;
SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;

typedef struct _SOCKET_ADDRESS

    LPSOCKADDR lpSockaddr;
    INT        iSockaddrLength;
    SOCKET_ADDRESS, »PSOCKET_ADDRESS, FAR    LPSOCKET_ADDRESS;
```

В поле *iAddressCount* возвращается количество адресных структур в списке, а поле *Address* — это массив адресов, специфичных для семейства протоколов.

В plug-and-play среде Win32 количество действительных адресов может изменяться динамически, поэтому приложениям не следует полагаться на постоянство информации, полученной от этой *ioctl*-команды. Так, приложения должны сначала вызвать *SIO\_ADDRESSLIST\_QUERY* для информации о текущих интерфейсах, а затем — *SIO\_ADDRESS\_UST\_CHANGE* для получения

уведомлений о будущих изменениях. Если список адресов изменится, то приложение должно повторить запрос.

Если размер предоставленного буфера не достаточен, то *WSAIocctl* вернет ошибку *WSAEFAULT*, а параметр *lebBytesReturned* — покажет требуемый размер буфера.

### Команда *SIO\_ADDRESS\_UST\_CHANGE*

Приложение использует эту команду для получения уведомления об изменениях в списке адресов локального транспорта для семейства протоколов данного сокета, с которым может связаться приложение. Функция *WSAIocctl* (версия Winsock 2+) не принимает никаких значений для ввода, и после ее успешного завершения параметры вывода не содержат никакой информации.

Существует несколько способов вызова этой команды. Если сокет — блокирующий, то вызов *WSAIocctl* не завершится до момента смены интерфейса. Если же сокет не в блокирующем режиме, вернется ошибка *WSAEWOULDBLOCK*. Затем приложение ожидает событий изменения маршрута, вызывая *WSAEventSelect* или *WSAAsyncSelect* с флагом *FDJROUTINGINTERFACEJCHANGE*, заданным в битовой маске сетевого события. Перекрытый ввод-вывод тоже можно использовать: предоставьте описатель события в структуре *WSAOVERLAPPED*, которая освободится после смены маршрутизации.

### Команда *SIOJGETINTERFACEJ.IST*

Эта ioctl-команда определена в *Ws2tcpip.h*. Она использует функцию *WSAIocctl* (версия Winsock 2+) для возврата информации о каждом интерфейсе на локальном компьютере. Вводить ничего не требуется, но по завершении возвращается массив структур *INTERFACEJINFO*:

```
typedef struct _INTERFACE_INFO

i   u_long          HFlags;           /* флаги интерфейса          */
    sockaddr_gen     iiAddress;        /* Адрес интерфейса          */
    sockaddr_gen     HBroadcastAddress; /* Широковещательный адрес */
    sockaddr_gen     iiNetmask;        /* Сетевая маска             */
> INTERFACE_INFO, FAR * LPINTERFACE_INFO;

#define IFFJJJP          0x00000001 /* Интерфейс работает      V
#define IFF.BROADCAST    0x00000002 /* Широковещание поддерживается */
#define IFF.LOOPBACK      0x00000004 /* Это петельный интерфейс   */
#define IFF_POINTTOPOINT  0x00000008 /* Это интерфейс "точка-точка" V
#define IFF_MULTICAST     0x00000010 /* Поддерживается многоадресная рассылка */

typedef union sockaddr_gen

    struct sockaddr Address;
    struct sockaddr_in AddressIn;
    struct sockaddr_in6 AddressIn6;
> sockaddr_gen;
```

Элемент *UFlags* возвращает битовую маску флагов, указывающих, рабочий ли это интерфейс (*JFFJIP*), а также поддерживается ли ширококовещание (*IFF^BROADCAST*) или многоадресная рассылка (*IFF MULTICAST*). Также видно, является ли интерфейс *петельным* (loopback) — *IFF LOOPBACK*, или служит для связи «точка-точка» — *IFFPOINTTOPOINT*. Три других поля содержат адрес интерфейса, адрес ширококовещания и соответствующую сетевую маску.

## loctl-команды Secure Socket Layer

Команды Secure Socket Layer (SSL) действуют только в Windows CE. В настоящий момент Windows 9x, NT и 2000 не содержат SSL-совместимого поставщика. Поэтому их поддерживает только Winsock 1.

### Команда **SOJSSL\_GET\_CAPABILITIES**

Функция *WSAIocctl* не имеет выходных значений. Команда *SO\_SSL\_GET\_CAPABILITIES* получает набор флагов, описывающих возможности поставщика защиты Windows Sockets. Буфер вывода должен быть указателем на битовое поле *DWORD*. На данный момент определен только флаг *SO\_CAP\_CUENT*.

### Команда **SO\_SSL\_GET\_FLAGS**

Функция *WSAIocctl* возвращает флаги, специфичные для 5-канала, связанного с сокетом. Она не имеет значений ввода, а буфер вывода должен быть указателем на битовое поле *DWORD*. Перечень допустимых флагов вы найдете в описании *SOJSLJETFLAGS*.

### Команда **SO\_SSL\_SET\_FLAGS**

Функция *WSAIocctl* не имеет выходных значений и задает флаги, специфичные для 5-канала, связанного с сокетом. Буфер ввода должен быть указателем на битовое поле *DWORD*. На данный момент определен только флаг *SSL\_FLAGJDEFERHANDSHAKE*, который позволяет приложениям отправлять и получать открытый текст до переключения на шифрованный. Этот флаг необходим для установки связи через прокси-серверы.

Как правило, поставщик защиты Windows Sockets выполняет безопасное согласование связи в рамках функции *connect*. Впрочем, если этот флаг включен, то согласование связи откладывается до момента, пока приложение не выдаст контрольный код *SO\_SSL\_PERFORM\_HANDSHAKE*. После согласования флаг сбрасывается.

### Команда **SO\_SSL\_GET\_PROTOCOLS**

Функция *WSAIocctl* не имеет выходных значений. Команда *SO\_SSL\_GET\_PROTOCOLS* получает список протоколов, которые поставщик поддерживает на этом сокете. Буфер вывода должен быть указателем на структуру *SSLPROTOCOLS*.

```
typedef struct _SSLPROTOCOL
{
    DWORD dwProtocol;
    DWORD dwVersion;
```

```

    DWORD dwFlags;
} SSLPROTOCOL, «LPSSLPROTOCOL;
typedef struct _SSLPROTOCOLS
{
    DWORD          dwCount;
    SSLPROTOCOL ProtocolList[1];
} SSLPPOTOCOLS, FAR .LPSSLPROTOCOLS;

```

Действительные протоколы для поля *dwProtocol*: *SSL\_PROTOCOL\_SSL2*, *SSL\_PROTOCOL\_SSL3* и *SSL\_PROTOCOL\_PCT1*.

### Команда **SO\_SSL\_SET\_PROTOCOLS**

Функция *WSAIoctl* не имеет выходных значений и задает список протоколов, которые должен поддерживать поставщик на этом сокете. Буфер ввода указывает на уже описанную структуру *SSLPROTOCOLS*.

### Команда **SO\_SSL\_SETJ/ALIDATE\_CERT\_HOOK**

Эта *ioctl*-команда задает для сокета указатель на функцию проверки для приема SSL-сертификатов. Функция *WSAIoctl* не имеет выходных параметров, это функция обратного вызова, применяемая поставщиком защиты Windows Sockets, когда он получает набора реквизитов от удаленной стороны. Буфер ввода должен быть указателем на структуру *SSLVALIDATECERTHOOK*:

```

typedef struct
{
    SSLVALIDATECERTFUNC   HookFunc;
    LPVOID                pvArg;
} SSLVALIDATECERTHOOK, «PSSLVALIDATECERTHOOK;

```

Поле *HookFunc* — указывает на функцию обратного вызова, проверяющую сертификат, *pvArg* — на данные, специфичные для приложения. Этот указатель приложения могут использовать в своих целях.

### Команда **SO\_SSL\_PERFORM\_HANDSHAKE**

Эта *ioctl*-команда инициирует надежное согласование на подключенном сокете, для которого перед соединением был задан флаг *SSL\_FLAG\_DEFER\_HANDSHAKE*. Буферы данных не требуются, но флаг *SSL\_FLAG\_DEFER\_HANDSHAKE* будет сброшен.

## ioctl-команды для ATM

Описанные в этом разделе *ioctl*-команды специфичны для семейства протоколов ATM. Они довольно просты и в основном касаются получения количества ATM-устройств и ATM-адресов локальных интерфейсов. Подробнее о механизмах адресации ATM — в главе 6.

### Команда **SIO\_GET\_NUMBER\_OF\_ATM\_DEVICES**

Функция *WSAIoctl* (версия Winsock 2+) не имеет значений для ввода. Команда *SIO\_GET\_NUMBER\_OF\_ATM\_DEVICES* заполняет буфер вывода, на который



ссылается *IpvOutBuffer*, данными типа *DWORD*, содержащими количество ATM-устройств в системе. Каждое устройство распознается по уникальному идентификатору, в диапазоне от 0 до  $n-1$ , где  $n$  — количество устройств, возвращенных этой *ioctl*-командой.

### Команда *SIO\_GET\_ATM\_ADDRESS*

С помощью этой команды функция *WSAIocctl* (версия Winsock 2+) получает локальный ATM-адрес, связанный с указанным устройством. Идентификатор устройства типа *DWORD* задается в буфере ввода, а буфер вывода, на который ссылается *IpvOutBuffer*, заполнен структурой *ATM\_ADDRESS*, содержащей локальный ATM-адрес, годный для использования с *bind*.

### Команда *SIO\_ASSOCIATE\_PVC*

Эта *ioctl*-команда с помощью функции *WSAIocctl* (версия Winsock 2+) связывает сокет с *постоянным виртуальным каналом связи* (permanent virtual circuit, PVC), указанным в буфере ввода, который содержит структуру *ATM\_PVCPARAMS*. Сокет должен быть из семейства адресов *AF\_ATM*. Параметров вывода нет. После успешного возврата из функции приложение способно начать отправку и прием данных, как если бы было установлено соединение.

Структура *ATM\_PVC\_PARAMS* определена таю

```
<
    ATM_CONNECTION_ID    PvcConnectionId;
    QOS                   PvcQos;
} ATM_PVC_PARAMS;

typedef struct
{
    DWORD    DeviceNumber;
    DWORD    VPI;
    DWORD    VCI;
} > ATM_CONNECTION_ID;
```

### Команда *SIO\_GET\_ATM\_CONNECTIONJD*

Обе функции (версия Winsock 2+) определяют, были ли считаны ООВ-данные. Команда *SIOjSET ATM\_CONNECTIONJD* получает идентификатор ATM-соединения, связанного с сокетом. После удачного возврата из этой функции буфер вывода, на который ссылается *IpvOutBuffer*, заполняется структурой *ATM\_CONNECTION\_ID*, содержащей номер устройства и значения VPI/VCI, которые были заданы ранее для *SIO ASSOCIATE\_PVC*.

## Резюме

Такое огромное разнообразие параметров сокета и `ioctl`-команд может на первый взгляд показаться излишним, но ведь они дают приложениям доступ к характеристикам, специфичным для протоколов, а также позволяют тонко настраивать приложения. В ряде случаев: например, с `AppleTalk` или `IrDA`, — приложение использует один или несколько параметров сокета или `ioctl`-команд. Но даже тогда, как правило, одновременно применяется лишь незначительная часть этих параметров. Конечно, недостаток в том, что не все параметры сокета и `ioctl`-команды доступны на всех платформах `Windows`. Это усложняет создание переносимых приложений.

•1, +

>•(•••

•н

# Регистрация и разрешение имен

В этой главе обсуждается независимая от протокола модель регистрации и разрешения имен, реализованная в Winsock 2. (Метод, использовавшийся в Winsock 1, устарел, и мы не будем его рассматривать.) Сначала мы обсудим общие принципы разрешения и регистрации имен, затем перейдем к различным моделям и описанию функций, предоставляемых протоколом Winsock 2 для разрешения имен. Мы также расскажем, как зарегистрировать службу, чтобы другие пользователи могли ее найти.

## Введение

Регистрация имен подразумевает сопоставление дружественного имени с зависимым от протокола адресом: например, имени узла с его IP-адресом. Большинству людей трудно запомнить такой адрес рабочей станции, как «157.54.185-186». Они предпочитают давать компьютерам более удобные для запоминания имена, например, «Ajones!». В протоколе IP соответствие IP-адресов именам обеспечивает служба Domain Name System (DNS). Другие протоколы предоставляют свои способы привязки используемых ими адресов к дружественным именам.

Требуется не только регистрировать и разрешать имена компьютеров, но также связывать адрес сервера Winsock, чтобы клиенты могли получать его для подключения к серверу. Например, сервер выполняется на порту номер 5000 компьютера с адресом 157.64.185.186. Если он всегда выполняется только на данной системе, можно жестко задать его адрес в клиентском приложении.

Но как быть, если необходим более динамичный сервер, способный выполняться на нескольких компьютерах — например, отказоустойчивое распределенное приложение? В случае отказа или перегруженности одного из серверов можно запустить экземпляр приложения на другой системе. Выяснить при этом, где же выполняются серверы, достаточно трудно. В идеале вам нужна возможность регистрировать свой *отказоустойчивый распределенный сервер* (Fault Tolerant Distributed Server) с несколькими адресами, а также динамически обновлять зарегистрированную службу и ее адреса.

Именно для этого и существуют регистрация и разрешение имен. В этой главе мы рассмотрим функции Winsock для регистрации и разрешения имен распределенного сервера.

## Модели пространства имен

Прежде всего рассмотрим различные модели пространства имен, используемые большинством протоколов. Пространство имен позволяет связать протокол и его атрибуты адресации с дружественным именем. К наиболее распространенным пространствам имен относятся DNS (используется протоколом IP) и NDS (NetWare Directory Services, используется протоколом IPX) компании Novell. Они сильно различаются как по организации, так и по реализации. Некоторые их свойства очень важны и помогают понять, как осуществляется регистрация и разрешение имен с помощью Winsock.

Существует три типа пространств имен: динамическое, статичное и постоянное. *Динамическое пространство имен* позволяет регистрировать сведения о службе на лету. Помимо прочего, это означает, что клиенты могут вести поиск службы в период выполнения. Обычно динамическое пространство имен периодически осуществляет широковебательную рассылку сведений о службе, извещая клиентов о ее постоянной доступности. В качестве примера динамических пространств имен можно назвать Service Advertising Protocol (SAP), который используется в средах Netware, и Name Binding Protocol (NBP) — применяется протоколом AppleTalk.

*Статичные пространства имен* — наименее гибкий из всех трех типов. Для регистрации службы в этом пространстве имен ее необходимо предварительно зарегистрировать вручную. Это означает, что зарегистрировать имя в статичном пространстве имен с помощью функций Winsock невозможно — Winsock включает лишь функции для разрешения имен. Пример статичного пространства имен — DNS, где вы вручную вводите IP-адреса и имена компьютеров в файл, который используется службой DNS для обработки запросов на разрешение имени.

*Постоянные пространства имен*, как и динамические, позволяют службам регистрировать сведения о себе на лету. Но в отличие от динамических, постоянные пространства хранят сведения о регистрации на энергонезависимых носителях, например, в файлах или на дисках. Постоянное пространство имен удаляет запись о службе только при получении от нее соответствующего запроса. Преимущества постоянного пространства — гибкость и отсутствие постоянной широковебательной рассылки сведений о доступности служб. Недостаток — если служба плохо написана, она может отключиться, не передав компоненту доступа (поставщику) к пространству имен запрос на удаление соответствующей записи, и клиенты будут получать ложную информацию о доступности службы. Пример постоянного пространства имен — NDS.

## Перечень пространств имен

Теперь рассмотрим, как получить список доступных на компьютере пространств имен. Большинство предопределенных пространств объявлены в заголовочном файле Nsapi.h. Каждому пространству имен присвоено целочисленное значение. Далее в списке перечислены некоторые распространенные пространства имен, имеющиеся на платформах Win32:

- К **NS\_SAP** — значение 1, пространство имен SAP, используемое в сетях IPX,
- III **NS\_NDS** — значение 2, пространство имен NDS, также используемое в сетях IPX,
- III **NS\_DNS** — значение 11, пространство имен DNS, широко используется в сетях TCP/IP и в Интернете,
- III **NS\_NTDS** — значение 32, пространство домена Windows NT, независимое от протокола, имеется в Windows 2000

Возвращаемый список пространств имен зависит от установленных на рабочей станции протоколов. Например, если на ней не установлен протокол IPX/SPX, при перечислении пространств имен система не вернет **NS\_SAP**.

Установив в системе пакет протоколов IPX/SPX, вы сможете лишь выполнять запросы к пространству имен SAP. Для регистрации собственных служб вам также потребуется служба SAP Agent. В некоторых случаях для корректного отображения адресов локального интерфейса IPX необходима служба Client Services for NetWare — без нее адреса отображаются состоящими из нулей. Кроме того, для работы с пространством имен NDS следует установить клиент NDS. Все указанные протоколы и службы можно установить при помощи Control Panel.

Функция *WsaEnumNameSpaceProviders* API Wmsock 2 позволяет программно получить список доступных в системе пространств имен.

```
INT WsaEnumNameSpaceProviders (
    LPDWORD lpdwBufferLength,
    LPWSANAMESPACE_INFO lpnspBuffer
),
```

Первый параметр — это размер буфера, ссылка на который передается в параметре *lpnspBuffer*. Буфер обычно представляет собой достаточно большой массив структур *WSANAMESPACEINFO*. Если при вызове функции размер буфера окажется недостаточным, произойдет сбой, параметру *lpdwBufferLength* будет присвоено значение, соответствующее минимально необходимому размеру буфера, и функция *WsaGetLastError* вернет *WSAEFAULT*. При любой ошибке функция возвращает *SOCKETERROR* или число возвращенных структур *WSANAMESPACEINFO*.

Структура *WSANAMESPACEINFO* описывает отдельное пространство имен, заданное в системе:

```
typedef struct _WSANAMESPACE_INFO {
    GUID NSProviderId,
    DWORD dwNameSpace,
    BOOL fActive,
    DWORD dwVersion,
    LPTSTR lpszIdentifier,
    > WSANAMESPACE_INFO *PWSANAMESPACE_INFO,
    LPWSANAMESPACE_INFO,
```

Фактически, существует два определения этой структуры — одно для UNICODE и одно для ANSI. Заголовочный файл Wmsock 2 выберет тип струк-

туры *WSANAMESPACEINFO* согласно способу сборки проекта. Все структуры, а также функции Winsock для регистрации и разрешения имен имеют как ANSI-, так и UNICODE-версию. Первый элемент структуры *WSANAMESPACE\_INFO* — *NSPromderId*. Это *глобально уникальный идентификатор* (globally unique identifier, GUID), описывающий данное конкретное пространство имен. Поле *dwNameSpace* — соответствующая пространству имен целочисленная константа: например, *NSJDNS* или *NS\_SAP*. *ЭлементJActive* — логическое значение, сообщающее о доступности пространства имен. Если оно равно TRUE, пространство имен доступно и может принимать запросы. Значение, равное FALSE, указывает, что поставщик пространства имен простаивает и не может принимать запросы, ссылающиеся непосредственно на него. Поле *dwVersion* указывает версию этого поставщика. Параметр *IpszIdentifier* — это строковый идентификатор, описывающий данный поставщик.

## Регистрация службы

Следующий этап — создать собственную службу, а также предоставить сведения о ней другим компьютерам сети и сделать ее доступной для них. Этот процесс называется регистрацией экземпляра службы в компоненте доступа к пространству имен. После регистрации вы вправе рассылать информацию о службе, кроме того, клиенты, которым требуется взаимодействовать со службой, смогут выполнять к ней запросы. Процесс регистрации службы включает два этапа. Первый этап — определение *класса службы* (service class), описывающего ее характеристики.

Важно различать класс службы и саму службу. Класс службы описывает пространства имен, в которых будет зарегистрирована служба, а также указывает определенные ее характеристики: например, требует она установки логического соединения или нет. Класс службы никак не описывает установление соединения клиентом. После регистрации класса службы можно зарегистрировать реальный экземпляр службы, ссылающийся на корректный класс службы, к которому он относится. Затем клиенты с помощью запросов узнают, где выполняется ваш экземпляр службы, и устанавливают соединение с ним.

## Определение класса службы

Регистрирует классы служб функция *WSAInstallServiceClass* из набора Winsock. INT *WSAInstallServiceClass* (*LPWASERVICECLASSINFO* lpServiceClassInfo),

Единственный параметр — *lpServiceClassInfo*, указывает на структуру *WSA-SERVICECLASSINFO*, определяющую атрибуты данного класса. Синтаксис структуры выглядит следующим образом:

```
typedef struct _WSAServiceClassInfo {
    LPGUID                lpServiceClassId,
    LPTSTR                lpszServiceClassName,
    DWORD                 dwCount,
    LPWANSCLASSINFO       lpClassInfos,
} > WASERVICECLASSINFO, *PWSERVICECLASSINFO, LPWASERVICECLASSINFO,
```

Первое поле — GUID, уникально идентифицирующий данный конкретный класс службы. Создать GUID, используемый в этом определении, можно несколькими способами. Прежде всего, при помощи утилиты Uuidgen.exe, но тогда для ссылки на этот GUID вам придется жестко задать его значение в каком-либо заголовочном файле. Здесь на помощь приходит второе решение. В заголовочном файле Svcguid.h несколько макросов создают GUID, на основе простого атрибута.

Например, определяя класс службы для SAP, который будет применяться для рассылки информации об IPX-приложении, вы можете воспользоваться макросом `SVCID^NETWARE`. Единственный параметр — идентификационный номер SAP, присваиваемый классу приложений. Число идентификаторов SAP в NetWare предопределено: 0x4 — для файловых серверов, 0x7 — для серверов печати и т.д. В этом случае необходим лишь легко запоминаемый идентификатор SAP, на основе которого создается GUID для соответствующего класса службы. Кроме того, существует несколько макросов, принимающих в качестве параметра номер порта, и возвращающих GUID соответствующей службы.

Рассмотрим заголовочный файл Svcguid.h, содержащий полезные макросы для выполнения обратной операции — получения номера порта службы по ее GUID. Перечислим наиболее часто используемые макросы для создания GUID на основе простых атрибутов протоколов, таких как номера портов или идентификаторы SAP. Эти макросы генерируют GUID, используя

- `SVCID TCP(Port)` — номер порта протокола TCP,
- III `SVCIDDNS(RecordType)` — тип записи DNS,
- K `SVCID_UDP(Port)` — номер порта протокола UDP,
- III `SVCIDNETWARE(Sapid)` — идентификатор SAP.

В заголовочном файле также есть константы для распространенных номеров портов, используемых службами типа FTP и Telnet.

Второе поле структуры `WSASERVICECLASSINFO` — `lpServiceClassName`, строка с именем данного конкретного класса службы. Два последних элемента взаимосвязаны. Поле `dwCount` ссылается на число структур `WSANSCLASSINFO`, переданное в поле `IpClassInfos`. Эти структуры определяют пространства имен и характеристики протоколов, которые распространятся на регистрирующиеся в данном классе службы. Структура `WSANSCLASSINFO` определена так:

```
typedef struct _WSANSClassInfo {
    LPSTR    lpzName,
    DWORD    dwNameSpace,
    \    DWORD    dwValueType,
    DWORD    dwValueSize,
    LPVOID    lpValue,
    >WSANSCLASSINFO, *PWSANSCLASSINFO, *LPWSANSCLASSINFO,
```

Поле `lpzName` определяет атрибут, принадлежащий классу службы. В табл. 10-1 перечислены возможные атрибуты, а также пространства имен, к которым обычно относятся различные типы служб. Тип значения всех ука-

занных атрибутов — *REG\_DWORD* Параметр *dwNameSpace* — пространство имен, на которое распространяется данный атрибут

**Табл 10-1. Классы служб**

Строковое значение	Константа определяет	Пространство имен	Описание
Sapid	<i>SERVICE_TYPE_VALUE_JAPNS_SAP</i>	<i>NS_SAP</i>	Идентификатор SAP
Connection-Oriented	<i>SERVICE_TYPE_VALUE_CONN</i>	<i>NS_CONN</i>	Указывает, требует ли служба установления соединения
TcpPort	<i>SERVICETYPEVALUE_TCP</i>	<i>NS_DNS</i>	Порт TCP
UdpPort	<i>SERVICETYPEVALUE_UDP</i>	<i>NS_NTDS</i>	Порт UDP

Последние три поля — *dwValueType*, *dwValueSize* и *IpValue*, описывают реальное значение, связанное с типом службы. Поле *dwValueType* указывает тип данных, связанный с этой записью, и может принимать одно из значений, используемых в реестре. Например, если значение равно *DWORD*, его тип — *REGDWORD*. Следующее поле — *dwValueSize*, содержит размер данных, переданных в параметре *IpValue*, представляющем собой указатель на данные.

Приведенный далее код определяет класс службы *Widget Server Class*

```
WSASERVICECLASSINFO    sci,
WSANSCLASSINFO          aNameSpaceClassInfo[4];
DWORD                   dwSapId = 200,
                        dwUdpPort = 5150,
                        dwZero = 0;
int                      ret;

memset(&sci, 0, sizeof(sci)),

SET_NETWORKWARE_SVCID(&sci.lpServiceClassId, dwSapId);
sci.lpszServiceClassName = (LPSTR)"Widget Server Class"
soi dwCount = 4,
sci lpClassInfos = aNameSpaceClassInfo;

memset(aNameSpaceClassInfo, 0, sizeof(WSANSCLASSINFO) * 4);
// Настройка пространства имен NTOS
aNameSpaceClassInfo[0].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[0].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[0].dwValueType = REG_DWORD;
aNameSpaceClassInfo[0].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[0].lpValue = &dwZero;

aNameSpaceClassInfo[1].lpszName = SERVICE_TYPE_VALUE_UDPPORT;
aNameSpaceClassInfo[1].dwNameSpace = NS_NTDS,
aNameSpaceClassInfo[1].dwValueType = REG_DWORD;
aNameSpaceClassInfo[1].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[1].lpValue = &dwUdpPort;
```



```
// Настройка пространства имен SAP
aNameSpaceClassInfo[2].lpzName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[2].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[2].dwValueType = REG_DWORD;
aNameSpaceClassInfo[2].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[2].lpValue = &dwZero;

aNameSpaceClassInfo[3].lpzName = SERVICE_TYPE_VALUE_SAPID;
aNameSpaceClassInfo[3].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[3].dwValueType = REG_0WORD;
aNameSpaceClassInfo[3].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[3].lpValue = idwSapId;

ret = WSAInstallServiceClass(&sci);
if (ret == SOCKET.ERROR)

    printf("WSAInstallServiceClass() failed %d\n", WSAGetLastError0);
```

Сначала выбирается GUID, под которым будет зарегистрирован класс. Все создаваемые службы относятся к классу Widget Server Class, описывающему общие атрибуты, которыми обладает экземпляр службы. В примере регистрируется класс службы с идентификатором NetWare SAP, равным 200. Это сделано лишь для удобства — мы могли бы выбрать произвольный GUID или даже GUID, основанный на номере порта UDP. Кроме того, служба может работать по протоколу UDP, который в нашем примере клиенты прослушивают на порте номер 5150.

Затем полю *dwCount* структуры *WSASERVICECLASSINFO* присваивается значение 4. В этом примере класс службы регистрируется и в пространстве имен SAP (*NS\_SAP*), и в пространстве домена Windows NT (*NS\_NTDS*). Обратите внимание: используются четыре структуры *WSANSCLASSINFO*, хотя класс регистрируется лишь в двух пространствах имен. Это связано с тем, что для каждого пространства имен определены два атрибута, которым необходимы отдельные структуры *WSANSCLASSINFO*. Мы также указываем, требует ли пространство имен установления логического соединения. В этом примере логическое соединение не нужно, поскольку мы присвоили полю *SERVICE\_TYPE\_VALUE\_CONN* логическое значение 0. Кроме того, для пространства домена Windows NT указан тип службы *SERVICE\_TYPE\_VALUE\_UDPPORT* и номер порта UDP, на котором обычно выполняется эта служба. Для пространства имен SAP идентификатор SAP задается использованием типа службы *SERVICE\_TYPE\_VALUE\_SAPID*.

Для каждой записи *WSANSCLASSINFO* необходимо определить идентификатор пространства имен, к которому относится данный тип службы; кроме того, следует задать тип и размер значения (табл. 10-1). В нашем примере все значения оказались *DWORD*. На последнем этапе вызывается функция *WSAInstallServiceClass* и ей в качестве параметра передается структура *WSASERVICECLASSINFO*. При успешном вызове функция вернет 0, в противном случае — *SOCKET\_ERROR*. Если структура *WSASERVICECLASSINFO* не действительна или некорректно сформирована, функция *WSAGetLastError* вернет *WSAEIN-*

*VAL*. При наличии такого же класса службы функция вернет *WSAEALREADY*. В нашем случае для удаления класса службы можно вызвать функцию *WSAEALREADY*:

```
INT WSARemoveServiceClass( LPGUID IpServiceClassId );
```

Единственный параметр функции — указатель на GUID, который определяет класс службы.

## Регистрация службы

После определения класса службы, описывающего ее общие атрибуты, можно зарегистрировать экземпляр службы и сделать ее доступной для других клиентов. Для регистрации службы применяется функция *WSASetService* из набора Winsock:

```
INT WSASetService (
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

Первый параметр — *lpqsRegInfo*, указатель на структуру *WSAQUERYSET*, определяющую конкретную службу. Параметр *essOperation* определяет выполняемое действие — регистрацию или удаление сведений о службе. Вот описание трех допустимых флагов.

**III RNRSERVICE REGISTER** — регистрирует службу Для поставщиков динамических имен это означает начало активной рассылки информации о службе, для поставщиков постоянных имен — обновление БД, для поставщиков статичных имен — ничего.

**III RNRSERVICE DEREGISTER** — удаляет все сведения о службе из реестра. Для поставщиков динамических имен это означает прекращение рассылки информации о службе, для поставщиков постоянных имен — удаление сведений о службе из БД, для поставщиков статичных имен — ничего.

**III RNRSERMCDELETE** — удаляет данный экземпляр службы из пространства имен. Служба, которую можно зарегистрировать, содержит несколько экземпляров (если при регистрации задан флаг *SERVICE\_MULTIPLE*), и данная команда удаляет лишь выбранный экземпляр службы (определяется структурой *CSADDRINFO*). Все сказанное также распространяется на поставщиков динамических и постоянных имен.

Третий параметр — *dwControlFlags*, равен 0 или флагу *SERVICE\_MULTIPLE*. Этот флаг используется, если данный экземпляр службы регистрируется с несколькими адресами. Например, требуется запустить службу на пяти системах. Структура *WSAQUERYSET*, переданная функции *WSASetService*, будет ссылаться на пять структур *CSADDRINFO*, каждая из которых описывает размещение одного из экземпляров службы. Для этого необходимо задать флаг *RNRSERVICE DELETE*. В табл. 10-2 приведены возможные комбинации управ-

ляющих флагов и флагов операций, а также описаны результаты выполнения команд в зависимости от того, существует уже служба или нет.

Табл. 10-2. Комбинации флагов *WSASetService*

Флаги		Служба существует	Службы не существует
<i>RNRSERVICE_</i> <i>REGISTER</i>	Не заданы	Перезаписать текущий экземпляр службы	Добавить по данному адресу новый экземпляр службы
	<i>SERVICE^MULTIPLE</i>	Обновить экземпляр службы, добавив новый адрес	Добавить по данному адресу новый экземпляр службы
<i>RNRSERVICE_</i> <i>DEREGISTER</i>	Не заданы	Удалить все экземпляры службы, но не сведения о ней (обычно <i>WSAQUERYSET</i> создается, однако число структур <i>CSADDRINFO</i> равно 0)	Ошибка, функция вернет <i>WSASERVICE_NOT_FOUND</i>
	<i>SERVICE^MULTIPLE</i>	Обновить службу, удалив указанный адрес. Сведения о службе не будут удалены, даже если не останется адресов.	Ошибка, функция вернет <i>WSASERVICE_NOT_FOUND</i>
<i>RNRSERVICE_</i> <i>DELETE</i>	Не заданы	Удалить из пространства имен все сведения о службе	Ошибка, функция вернет <i>WSASERVICE_NOT_FOUND</i>
	<i>SERVICE^MULTIPLE</i>	Обновить службу, удалив указанный адрес. Если адресов не останется, сведения о службе будут удалены.	Ошибка, функция вернет <i>WSASERVICE_NOT_FOUND</i>

Теперь рассмотрим структуру *WSAQUERYSET* — ее необходимо заполнить и передать функции *WSASetService*:

```
typedef struct _WSAQUERYSET {
    DWORD           dwSize;
    LPTSTR          lpszServiceInstanceName
    LPGUID          lpServiceClassId;
    LPWSAVERSION    lpVersion;
    LPTSTR          lpszComment;
    DWORD           dwNameSpace;
    LPGUID          lpNSProviderId;
    LPTSTR          lpszContext;
    DWORD           dwNumberOfProtocols;
    1PAFPROTOCOLS  lpafpProtocols;
    LPTSTR          lpszQueryString;
    DWORD           dwNumberOfCsAddrs;
    LPCSADDR.INFO   lpcsaBuffer;
    DWORD           dwOutputFlags;
```

```

LPBLOB          IpBlob;
} WSAQUERYSET, *PWSAQUERYSET, *LPWSAQUERYSET;

```

Полю *dwSize* следует присвоить значение, соответствующее размеру структуры *WSAQUERYSET*. Поле *ipzServiceInstanceName* содержит строковый идентификатор, задающий имя данного экземпляра сервера. Поле *ipServiceClassId* — GUID класса, к которому принадлежит данный экземпляр службы. Поле *ipVersion* является необязательным. Оно позволяет указать сведения о версии, которые могут оказаться полезными клиенту при запросе к службе. Поле *ipzComntnent* также является необязательным и позволяет задать любой строковый комментарий. Поле *dwNameSpace* указывает пространства имен, в которых будет зарегистрирована служба.

При работе с одним пространством имен укажите только одно значение, в противном случае присвойте полю значение *NS\_ALL*. Кроме того, можно сослаться на собственный поставщик пространства имен (подробней о создании собственного пространства имен — в главе 14). Для собственного поставщика пространства имен полю *dwNameSpace* присваивается значение 0, а поле *ipNSProviderId* задает GUID, представляющий собственный поставщик. Поле *ipzContext* указывает начальную точку запроса в иерархичном пространстве имен, например, DNS.

Поля *dwNumberOfProtocols* и *ipaIpProtocols* — дополнительные параметры, позволяющие сузить поиск и вернуть сведения только о необходимых протоколах. Поле *dwNumberOfProtocols* ссылается на число структур *AFPROTOCOLS*, имеющихся в массиве *ipafpProtocols*. Синтаксис структуры *AFPROTOCOLS* следующий:

```

typedef struct AFPROTOCOLS {
    INT iAddressFamily;
    INT IProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;

```

Первое поле — *iAddressFamily*, это константа семейства адресов, например, *AFINET* или *AF\_IPX*. Второе поле — *iProtocol*, протокол из данного семейства адресов, например *IPPROTOJTCP* или *NSPROTOJPX*.

Следующее поле структуры *WSAQUERYSET* — *ipzQueryString*, является необязательным и используется только теми пространствами имен, которые поддерживают расширенные SQL-запросы, например, Whois++. Это поле задает строку расширенного запроса.

Два следующих поля наиболее важны при регистрации службы. Поле *dwNumberOfCsAddrs* указывает число структур *CSADDRINFO*, переданных в параметре *ipcsaBuffer*. Структура *CSADDRINFO* определяет семейство адресов и фактический адрес, по которому находится служба. Если имеется несколько структур, будут доступны несколько экземпляров службы. Синтаксис структуры *CSADDRINFO* следующий:

```

typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS LocalAddr;
    SOCKET_ADDRESS RemoteAddr;
    INT             iSocketType;
    INT             iProtocol;
}

```

```
} CSADDR_INFO,

typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr,
    INT         iSockaddrLength,
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR    LPSOCKET_ADDRESS,
```

Сюда также включено определение *SOCKET\_ADDRESS* При регистрации службы вы можете задать локальные и удаленные адреса Поле локального адреса (*LocalAddr*) позволяет указать адрес, с которым должен связываться экземпляр данной службы, поле удаленного адреса (*RemoteAddr*) — адрес, который клиент должен использовать при вызовах *connect* и *sendto* Два других поля указывают тип сокета (например, *SOCK\_STREAM* или *SOCK\_DGRAM*) и семейство протоколов (например, *AF\_INET* или *AF\_IPX*) для данного адреса

Последние два поля структуры *WSAQUERYSET* — *dwOutputFlags* и *ipBlob* При регистрации службы они обычно не требуются, и применяются в запросе к экземпляру службы Структура *BLOB* возвращает лишь поставщик пространства имен, то есть при регистрации службы нельзя добавить собственную структуру *BLOB*, которая будет возвращаться в клиентских запросах

В табл. 10-3 перечислены поля структуры *WSAQUERYSET*, и указано, какие из них являются обязательными, а какие нет — в зависимости от выполняемого действия запроса или регистрации службы

**Табл. 10-3. Поля *WSAQUERYSET***

Поле	Запрос	Регистрация
<i>dwSize</i>	Обязательное	Обязательное
<i>ipSzServiceInstanceName</i>	Требуется строка или «».	Обязательное
<i>ipServiceClassId</i>	Обязательное	Обязательное
<i>ipVersion</i>	Необязательное	Необязательное
<i>ipSzComment</i>	Игнорируется	Необязательное
<i>dwNameSpace</i> , <i>ipNSProviderId</i>	Должно быть указано одно из этих полей	Должно быть указано одно из этих полей
<i>ipSzContext</i>	Необязательное	Необязательное
<i>dwNumberOfProtocols</i>	О или больше	О или больше
<i>ipafpProtocols</i>	Необязательное	Необязательное      КО
<i>ipSzQuerySnnng</i>	Необязательное	Игнорируется
<i>dwNumberOfCsAddrs</i>	Игнорируется	Обязательное
<i>ipcsaBuffer</i>	Игнорируется	Обязательное
<i>dwOutputFlags</i>	Игнорируется	Необязательное
<i>IpBlob</i>	Игнорируется, может возвращаться запросом	Игнорируется

**Пример регистрации службы**

А теперь рассмотрим на примере, как зарегистрировать собственную службу в пространствах имен SAP и NTDS одновременно Пространство домена Windows NT предоставляет достаточно мощные возможности Тем не менее,

кое-что следует помнить. Во-первых, пространство домена Windows NT требует Windows 2000, поскольку оно основано на Active Directory. Это также означает, что у рабочей станции Windows 2000, на которой вы собираетесь зарегистрировать и (или) искать службу, должна иметься учетная запись в данном домене, в противном случае система не сможет обратиться к Active Directory.

Кроме того, пространство домена Windows NT может регистрировать адреса сокетов из любого семейства протоколов. Это означает, что все IP- и IPX-службы можно регистрировать в одном пространстве имен, а удаление и добавление IP-адресов — осуществлять динамически. Для простоты из кода исключен контроль ошибок.

#### Листинг 10-1. Пример функции *WSASetService*

```

SOCKET          socks[2],
WSAQUERYSET     qs
CSADDR_INFO     lpCSAddr[2],
SOCKADDR_IN     sa_in,
SOCKADDR_IPX    sa_ipx,
IPX_ADDRESS_DATA ipx_data,
GUID            guid = SVCID_NETWARE(200),
int             ret, cb,

memset(&qs, 0, sizeof(WSAQUERYSET)),
qs dwSize = sizeof(WSAQUERYSET),
qs lpszServiceInstanceName = (LPSTR) Widget Server ,
qs lpServiceClassId = &guid,
qs dwNameSpace = NS_ALL,
qs lpNSProviderId = NULL,
qs lpCSaBuffer = lpCSAddr,
qs lpBlob = NULL,

// Задаем IP-адрес нашей службы

memset(&sa_in, 0, sizeof(sa_m)),
sa_in sin_family = AF_INET,
sa_m sin_addr s_addr = htonl(INADDR_ANY),
sa_m sin_port = 5150,

socks[0] = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP),
ret = bmd(socks[0], (SOCKADDR *)&sa_m, sizeof(sa_in)),

cb = sizeof(sa_in),
getsockname(socks[0], (SOCKADDR *)&sa_m, &cb),

lpCSAddr[0] iSocketType = SOCK_DGRAM,
lpCSAddr[0] lProtocol = IPPROTO_UDP,
lpCSAddr[0] LocalAddr lpSockaddr = (SOCKADDR *)&sa_in,
lpCSAddr[0] LocalAddr iSockaddrLength = sizeof(sa_in),
lpCSAddr[0] RemoteAddr lpSockaddr = (SOCKADDR *)&sa_in,
lpCSAddr[0] RemoteAddr iSockaddrLength = sizeof(sa_in),

```

**Листинг 10-1.**    *{продолжение}*

””  
“”

//

II Задаем IPX-адрес нашей службы  
//

```
memset(sa_ipx.sa_netnum, 0, sizeof(sa_ipx.sa_netnum));
memset(sa_ipx.sa_nodenum, 0, sizeof(sa_ipx.sa_nodenum));
sa_ipx.sa_family = AF_IPX;
sa_ipx.sa_socket = 0;
```

```
socks[1] = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);
```

```
ret = bind(socks[1], (SOCKADDR *)&sa_ipx, sizeof(sa_ipx));
```

```
cb = sizeof(IPX_ADDRESS_DATA);
memset (&ipx_data, 0, cb);
ipx_data.adapternum = 0;
```

```
ret = getsockopt(socks[1], NSPROTO_IPX, IPX_ADDRESS,
(char *)&ipx_data, &cb);
```

```
cb = sizeof(SOCKADDR_IPX);
getsockname(socks[1], (SOCKADDR *)&sa_ipx, &cb);
```

```
memcpy(sa_ipx.sa_netnum, ipx_data.netnum, sizeof(sa_ipx.sa_netnum));
memcpy(sa_ipx.sa_nodenum, ipx_data.nodenum, sizeof(sa_ipx.sa_nodenum));
```

er  
\*?

```
lpCSAddr[1].iSocketType = SOCK_DGRAM;
lpCSAddr[1].iProtocol = NSPROTO_IPX;
lpCSAddr[1].LocalAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].LocalAddr.iSockaddrLength = sizeof(sa_ipx);
lpCSAddr[1].RemoteAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].RemoteAddr.iSockaddrLength = sizeof(sa_ipx);
```

^  
P  
P  
\*  
ч

```
qs.dwNumberOfCsAddrs = 2;
```

V ,

```
ret = WSASetService(&qs, RNRSERVICE.REGISTER, 0L);
```

I n> \*H I

В листинге 10-1 показано, как настроить экземпляр службы, чтобы ее клиент мог найти адреса, необходимые для взаимодействия с ней. Прежде всего, следует инициализировать структуру *WSAQUERYSET*. Кроме того, необходимо задать имя экземпляра службы, назовем его Widget Server. Другой важный шаг — использовать тот же GUID, который применялся для регистрации нашего класса службы. Здесь мы работаем с классом Widget Service Class (определение дано в предыдущем разделе), GUID которого — *SVCID\_NETWARE(200)*. Следующий этап — задать интересующие нас пространства имен. Наша служба выполняется по протоколам IPX и UDP, и поэтому мы указываем *NS\_ALL*. Поскольку мы задаем уже существующее пространство имен, присвоим параметру *ipNSProviderId* значение *NULL*.

Затем следует настроить структуры *SOCKADDR* в массиве *CSADDRINFO*, которые функция *WSASetService* передает в качестве поля *ipcsaBuffer* структу-

ры *WSAQUERYSET*. Как видите, перед настройкой структуры *SOCKADDR* мы действительно создаем сокеты и связываем их с локальным адресом. Это обусловлено тем, что нам необходимо узнать точный локальный адрес, к которому будут подключаться клиенты. Например, мы связываем создаваемый для сервера UDP-сокет с *INADDR\_ANY*, таким образом, получить реальный IP-адрес без вызова функции *getsockname* невозможно. На основе полученной от функции *getsockname* информации можно создать структуру *SOCKADDRIN*. В структуре *CSADDR\_INFO* задаем тип и протокол сокета. Два других поля содержат локальный (с которым должен быть связан сервер) и удаленный адрес (который клиент будет использовать для подключения к службе).

Следующий шаг — настроить службу, выполняющуюся по протоколу IPX. Из материалов главы 6 вы знаете, что серверы должны быть связаны с номером внутренней сети, для этого номер сети и узла должны быть нулевыми. Опять же, таким образом вы не сможете получить необходимый клиентам адрес. Чтобы получить его, вызовите параметр сокета *IPX\_ADDRESS*. Заполняя структуру *CSADDRINFO* для протокола IPX, укажите *SOCK\_DGRAM* и *NSPROTO\_IPX* в качестве типа сокета и протокола соответственно.

Завершающий этап — присвоить полю *dwNumberOfCsAddrs* структуры *WSAQUERYSET* значение 2, поскольку для установления соединения клиенты могут использовать два адреса — UDP и IPX. Затем вызовите функцию *WSASetService*, передав ей структуру *WSAQUERYSET*, флаг *RNRSERVICE\_REGISTER* и не передавая управляющих флагов. Управляющий флаг *SERVICE\_MULTIPLE* не указывается, чтобы при удалении сведений о службе удалялась информация обо всех ее экземплярах (IPX- и UDP-адреса).

В приведенном примере не учитывается один случай: компьютеры с несколькими сетевыми адаптерами. Если вы создадите сервер на основе протокола UDP, привязывающийся *KINADDRANYHZL* компьютерам с несколькими сетевыми адаптерами, клиент сможет подключаться к этому серверу, используя любой из доступных интерфейсов. В протоколе IP функции *getsockname* недостаточно: вам потребуется получить все локальные IP-адреса. Это можно осуществить несколькими способами, в зависимости от платформы, на которой вы работаете. Один из распространенных методов, применимый на всех платформах — вызвать функцию *gethostbyname*, которая вернет список IP-адресов для имени. В Winsock можно также вызвать ioctl-команду *SIOGETINTERFACELIST*. Для Windows 2000 доступна ioctl-команда *SIO\_ADDRESSLIST\_QUERY*.

Кроме того, можно воспользоваться функциями IP helper (приложение B). Простое разрешение имен TCP/IP и функция обсуждаются в главе 6, а команды ioctl — в главе 9. На прилагаемом компакт-диске содержится пример (файл *Rnrcs.c*), в котором реализована работа с компьютерами с несколькими сетевыми адаптерами.

## Запрос к службе

Теперь рассмотрим, как клиент может запросить пространство имен для данной службы и получить информацию, необходимую для установления связи. Разрешение имен несколько проще, чем регистрация служб. Для выпол-



нения запросов используются три функции: *WSALookupServiceBegin*, *WSALookupServiceNext* и *WSALookupServiceEnd*.

Первый этап — вызвать функцию *WSALookupServiceBegin*, которая иницирует запрос, задавая ограничения для его выполнения:

```
INT WSAStartup (
    LPWSAQUERYSET lpqsRestrictions,
    DWORD dwControlFlags,
    HANDLE lphLookup
);
```

Первый параметр — структура *WSAQUERYSET*, ограничивающая запрос, например в части количества опрашиваемых пространств имен. Второй параметр — *dwControlFlags*, определяет глубину поиска. Модель поведения запроса и то, какие данные он вернет, определяют следующие флаги.

**II LUPDEEP** — в иерархичных пространствах имен задает глубину запроса по отношению к первому уровню.

**III LUPCONTAINERS** — вернуть только объекты-контейнеры. Этот флаг действителен лишь в иерархичных пространствах имен.

**III LUPNOCONTAINERS** — не возвращать какие-либо контейнеры. Флаг также действителен лишь в иерархичных пространствах имен.

- **LUPFLUSHCACHE** — игнорировать кэш и опрашивать непосредственно пространство имен. Заметьте: не все поставщики пространств имен кэшируют запросы.

**III LUPFLUSHPREVIOUS** — указать поставщику пространства имен отбросить ранее возвращенный набор сведений. Обычно используется после того, как *WSALookupServiceNext* вернет *WSA\_NOT\_ENOUGH\_MEMORY*. Набор, не помещающийся в предоставленный буфер, отбрасывается, после чего извлекается следующий.

- **LUPNEAREST** — вернуть результаты, упорядочив их по расстоянию. Мера расстояния определяется поставщиком имен, поскольку при регистрации службы соответствующие сведения не указываются. Поставщикам имен не требуется поддерживать данную концепцию.
- **LUPRESSERVICE** — указывает, что локальные адреса должны быть возвращены в структуре *CSADDRINFO*.

**U LUP\_RETURN\_ADDR** — вернуть адреса как *ipcsaBuffer*.

- **LUP\_RETURN\_ALIASES** — получить только сведения о псевдонимах. Каждый псевдоним будет возвращаться при успешных вызовах функции *WSALookupServiceNext* и для него будет задан флаг *RESULT JS\_AUAS*.

**III LUP\_RETURN\_ALL** — вернуть все доступные сведения.

**III LUPRETURNBLOB** — вернуть все частные данные как *ipBlob*.

**III LUPRETURNCOMMENT** — вернуть комментарий как *ipszComment*.

**Ж LUPRETURNNAME** — вернуть имя как *ipszServiceInstanceName*.

**III LUP\_RETURN\_TYPE** — вернуть тип как *ipServiceClassId*.

**Ж LUP\_RETURN\_VERSION** — вернуть версию как *ipVersion*,

Последний параметр имеет тип *HANDLE* и инициализируется при возвращении функции. В случае успеха возвращенное значение равно 0, иначе — *SOCKETJERROR*. Если один или несколько параметров не действительны, функция *WSAGetLastError* возвращает *WSAEINVAL*. Если имя найдено в пространстве имен, но заданным ограничениям не соответствуют какие-либо данные, будет возвращен код ошибки *WSANOJDATA*. Если службы не существует, функция *WSAGetLastError* возвращает *WSAEINVAL*.

После вызова функция возвращает дескриптор *WSALookupServiceBegin*, который передается функции *WSALookupServiceNext*, возвращающей информацию:

```
INT WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
```

Функция *WSALookupServiceBegin* возвращает дескриптор *hLookup*. Параметр *dwControlFlags* имеет в функции *WSALookupServiceBegin* то же самое значение, но поддерживается лишь *LUP\_FLUSHPREVIOUS*. Параметр *lpdwBufferLength* — длина буфера, переданного в качестве *lpqsResults*. Поскольку структура *WSAQUERYSET* может содержать данные больших двоичных объектов, часто требуется передать буфер, объем которого превосходит объем структуры. Если размер буфера не достаточен для возвращенных данных, произойдет сбой и функция вернет *WSA\_NOT\_ENOUGH\_MEMORY*.

Инициировав запрос с помощью функции *WSALookupServiceBegin*, вызывайте функцию *WSALookupServiceNext* до тех пор, пока система не выдаст сообщение об ошибке *WSA\_E\_NO\_MORE* (10110). Помните, что в предыдущих версиях Winsock при отсутствии данных возвращалась ошибка *WSAENOMORE* (10102), поэтому надежное приложение должно проверять оба кода. Получив все данные или завершив опрос, вызовите функцию *WSALookupServiceEnd*, передав ей использовавшуюся в запросах переменную *HANDLE*:

```
INT WSALookupServiceEnd ( HANDLE hLookup );
```

## Создание запроса

Рассмотрим, как опросить зарегистрированную в предыдущем разделе службу. Прежде всего, необходимо настроить структуру *WSAQUERYSET*, определяющую запрос:

```
WSAQUERYSET qs;
DWORD fluid = SCD_NEWWARE(200);
AFPROTOCOLS afp[2] * {{AF_IPX, NSPROTO.IPX}, {AF_INET, IPPROTOJUDP}};
HANDLE hLookup;
int ret;

memset(&qs, 0, sizeof(qs));
qs.dwSlze = sizeof (WSAQUERYSET);
```

```

qs.lpszServiceInstanceName = "Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.dwNumberOfProtocols = 2;
qs.lpszProtocols = afp;

```

```

ret = WSALookupServiceBegin(&qs, LUP_RETURN_ADDR | LUP_RETURN_NAME,
    &hLookup);
if (ret == SOCKET_ERROR)
    // Ошибка

```

Помните, что все операции поиска служб основаны на GUID класса службы, на котором построена искомая служба. Переменной *guid* присваивается идентификатор класса службы сервера. Сначала вы инициализируете переменную *qs* со значением 0 и сохраняете в поле *dwSize* размер структуры. Следующий шаг — указать имя искомой службы. Это может быть как точное имя, так и звездочка (\*); в последнем случае функция вернет все службы с данным GUID класса службы. Далее с помощью константы *NS\_ALL* указано, что поиск должен вестись во всех пространствах имен. Последний этап — настройка протоколов, по которым может соединяться клиент, в нашем случае это IPX и UDP/IP. Для этого используется массив из двух структур *AFPROTOCOLS*.

Теперь можно начать опрос: вызовите функцию *WSALookupServiceBegin*. Первый параметр — структура *WSAQUERYSET*, последующие — флаги, определяющие, какие данные будут возвращены при обнаружении искомой службы. Здесь вы указываете, что требуются сведения об адресах и имя службы; для этого создается логическое условие «ИЛИ» из флагов *LUP\_RETURN\_ADDR* и *LUP\_RETURN\_NAME*. Флаг *LUP\_RETURN\_NAME* необходим, только если в качестве имени службы вы указали звездочку (\*), в противном случае имя службы уже известно. Последний параметр — переменная *HANDLE*, идентифицирующая данный конкретный запрос. Она инициализируется при успешном возвращении данных.

После успешного открытия запроса вызывайте функцию *WSALookupServiceNext*, пока не будет возвращен код *WSA\_E\_NO\_MORE*. При каждом успешном вызове функции возвращаются сведения о службе, соответствующие заданным критериям:

```

char          buff[sizeof(WSAQUERYSET) + 2000];
DWORD         dwLength, dwErr;
WSAQUERYSET  *pqs = NULL;
SOCKADDR     *addr;
int          i;

pqs = (WSAQUERYSET *)buff;
dwLength = sizeof(WSAQUERYSET) + 2000;
while (1)
{
    ret = WSALookupServiceNext(hLookup, 0, &dwLength, pqs);
    if (ret == SOCKET_ERROR)

```

```

if ((dwErr = WSAGetLastErrorO) == WSAEFAULT)
{
    printf("Buffer too small; required size is: %d\n", dwLength);
    break;
}
else if ((dwErr == WSAENOMORE) || (dwErr == WSAENOMORE))
{
    break;
}
else
{
    printf("Failed with error: %d\n", dwErr);
    break;
}
}

for (i = 0; i < pqs->dwNumberOfCsAddrs; i++)
{
    addr = (SOCKADDR *)pqs->lpcsaBuffer[i].RemoteAddr.lpSockaddr;
    if (addr->sa_family == AF_INET)
    {
        SOCKADDR_IN *ipaddr = (SOCKADDR_IN *)addr;
        printf("IP address:port = %s:%d\n", inet_ntoa(addr->sin_addr),
            addr->sin_port);
    }
    else if (addr->sa_family == AF_IPX)
    {
        SOCKADDR_IPX *ipxaddr = (SOCKADDR_IPX *)addr;
        printf("X02XX02XX02XX02X.X02XX02XX02XX02XX02XX02X:X04X",
            (unsigned char)ipxaddr->sa_netnum[0],
            (unsigned char)ipxaddr->sa_netnum[1],
            (unsigned char)ipxaddr->sa_netnum[2],
            (unsigned char)ipxaddr->sa_netnum[3],
            (unsigned char)ipxaddr->sa_nodenum[0],
            (unsigned char)ipxaddr->sa_nodenum[1],
            (unsigned char)ipxaddr->sa_nodenum[2],
            (unsigned char)ipxaddr->sa_nodenum[3],
            (unsigned char)ipxaddr->sa_nodenum[4],
            (unsigned char)ipxaddr->sa_nodenum[5],
            ntohs(ipxaddr->sa_socket));
    }
}

```

WSALookupServiceEnd(hLookup);

Этот код вполне понятен, хотя и упрощен. При вызове функции *WSALookupServiceNext* требуются только действительный дескриптор запроса, непосредственно возвращаемый буфер и его длина. Указывать какие-либо управляющие флаги не нужно, поскольку единственный допустимый флаг данной функции — *LUPJFLUSHPREVIOUS*. Если размер переданного буфера не достаточен, и данный флаг задан, результаты вызова функции отбрасываются. Тем не менее, в примере флаг *LUP\_FLUSHPREVIOUS* не используется, и поэтому при недостаточном размере буфера генерируется ошибка *WSAEFAULT*.

В этом случае параметру *ipdwBufferLength* присваивается значение, соответствующее требуемому размеру буфера. В примере используется буфер с фиксированным размером, равным размеру структуры *WSAQUERYSET* плюс 2000 байт. Поскольку вам необходимы только имена и адреса службы, такого размера должно хватить. Конечно, серьезные приложения должны уметь обрабатывать ошибку *WSAEFAULT*.

После успешного вызова функции *WSALookupServiceNext* в буфер *WSAQUERYSET* помещается структура, содержащая результаты. В запросе требовалось получить имена и адреса, и поэтому наиболее интересные для нас поля структуры *WSAQUERYSET* — это *ipszServiceInstanceName* и *ipcsaBuffer*. Поле *IpszServiceInstanceName* содержит имя службы, а поле *IpcsaBuffer* — представляет массив структур *CSADDRINFO* с ее адресами. Параметр *dwNumberOfCsAddrs* указывает, сколько адресов возвращено. В коде примера мы просто выводим все адреса. Убедитесь, что будут выводиться только IP- и IPX-адреса, поскольку это единственные семейства адресов, указанные при открытии запроса.

Если в запросе в качестве имени службы указана звездочка (\*), при каждом вызове функции будет возвращен конкретный экземпляр этой службы, выполняющийся на одном из компьютеров сети (конечно, при условии, что несколько экземпляров действительно зарегистрированы и выполняются). После того, как функция *WSA\_E\_NO\_MORE* вернет все экземпляры службы, будет сгенерирована ошибка и цикл прекратится. Последнее, что необходимо сделать — вызвать функцию *WSALookupServiceEnd*, передав ей дескриптор запроса. При этом будут освобождены все выделенные запросу ресурсы.

## Запрос к DNS

Как уже упоминалось, пространство имен DNS статично, то есть не позволяет динамически зарегистрировать собственную службу. Тем не менее, для запросов к DNS можно воспользоваться функциями разрешения имен Winsock.

Выполнить DNS-запрос сложнее, чем обычный запрос о наличии зарегистрированной службы, поскольку поставщик пространства имен DNS возвращает информацию в форме BLOB. Почему? В главе 6, где обсуждалась функция *gethostbyname*, мы говорили, что при поиске по имени возвращается структура *HOSTENT*, содержащая не только IP-адреса, но и их псевдонимы. Зачастую эта информация не умещается в поля структуры *WSAQUERYSET*.

Формат BLOB-данных плохо документирован, поэтому для прямых запросов к DNS приходится изобретать обходные пути. Прежде всего рассмотрим, как открыть запрос. В файле *Dnsquery.c* на прилагаемом компакт-диске содержится полный код для прямого запроса к DNS, а здесь мы рассмотрим его поэтапно. Следующий код инициализирует DNS-запрос:

```
WSAQUERYSET qs;
AFPROTOCOLS afp [2] = {{AF_INET, IPPROTO_TCP}, {AF_INET, IPPROTO_UDP}};
GUID hostnameeguid = SVCID_INET_HOSTADDRBYNAME,
DWORD dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
HANDLE hQuery;
```

```

qs = (WSAQUERYSET *)buff;
memset(&qs, 0, sizeof(qs));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = argv[1];
qs.lpServiceClassId = ihostnameeguid;
qs.dwNameSpace = NS_DNS;
qs.dwNumberOfProtocols = 2;
qs.lpfafProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_NAME | LUP_RETURN_BLOB,
    ihQuery);
if (ret == SOCKET_ERROR)
    // Ошибка

```

Настройка запроса осуществляется почти так же, как и в предыдущем примере. Наиболее значительное отличие — используется предопределенный GUID *SVCIDJNETJiOSTADDRBYNAME*, он идентифицирует запросы имен компьютеров. Параметр *lpszServiceInstanceName* — имя компьютера, которое требуется разрешить. Поскольку имена разрешаются с использованием DNS, в качестве параметра *dwNameSpace* следует передать лишь *NSJDNS*. Наконец, в параметре *lpaJProtocols* передается массив из двух структур *AFPROTOCOLS*, которые определяют используемые запросом протоколы TCP/IP и UDP/IP.

После создания запроса вызовите функцию *WSALookupServiceNext*, чтобы вернуть данные

```

char        buff[sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048)];
DWORD       dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
WSAQUERYSET *pqs;
HOSTENT     *hostent;

pqs = (WSAQUERYSET *)buff;
pqs->dwSize = sizeof(WSAQUERYSET);
ret = WSALookupServiceNext(hQuery, 0, idwLength, pqs);
if (ret == SOCKET_ERROR)
    // Ошибка

WSALookupServiceEnd(hQuery);

hostent = pqs->lpBlob->pBlobData;

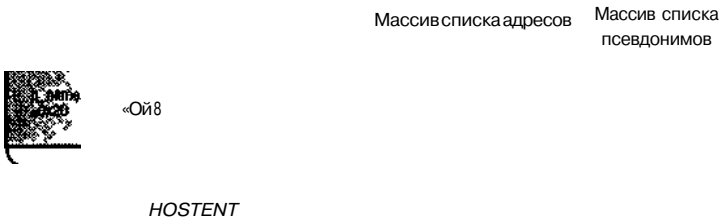
```

Поскольку поставщик пространства имен DNS возвращает сведения о компьютере в форме BLOB, необходимо выделить буфер достаточного размера. Именно поэтому используется буфер, равный по объему сумме «структура *WSAQUERYSET* + структура *HOSTENT* + дополнительные 2048 байт». Если и такого размера окажется не достаточно, при вызове функции произойдет сбой и будет возвращено значение *WSAEFAULT*. В DNS-запросе все сведения о компьютере возвращаются в структуре *HOSTENT*, даже если имя компьютера связано с несколькими IP-адресами. Таким образом, не требуется многократно вызывать функцию *WSALookupServiceNext*.

Теперь наступает самый сложный этап — расшифровка BLOB, возвращенного запросом. Из главы 6 вы знаете, что структура *HOSTENT* определена так

```
typedef struct hostent {
    char FAR * h_name,
    char FAR * FAR * h_aliases,
    short h_addrtype,
    short h_length,
    char FAR * FAR * h_addr_list,
} HOSTENT;
```

Когда структура *HOSTENT* возвращается в виде BLOB-данных, указатели внутри нее представляют собой смещения по адресам памяти, где хранятся данные. Смещение отсчитывается от начала BLOB-данных. В связи с этим для доступа к данным требуется зафиксировать указатели и сделать так, чтобы они ссылались на абсолютные адреса памяти. На рис. 10-1 изображена структура *HOSTENT* и карта возвращенной памяти.



**Рис. 10-1. BLOB-формат структуры *HOSTENT***

DNS-запрос выполняется по имени компьютера Riven, связанного с одним IP-адресом и не имеющего псевдонимов. У каждого поля структуры имеется значение смещения. Чтобы поля ссылались на действительное расположение данных, необходимо прибавить значение смещения к адресу в заголовке структуры *HOSTENT*. Подобную операцию следует выполнить для полей *h\_name*, *h\_aliases* и *h\_addrlist*. Кроме того, поля *h\_aliases* и *h\_addrlist* представляют собой массивы указателей.

Итак, получен верный указатель на массив указателей: каждое 32-битное поле в этом массиве ссылается на область памяти, содержащую указатели. В поле *h\_addr\_list* (рис. 10-1) начальное смещение составляет 16 байт — это ссылка на байт, следующий за структурой *HOSTENT*, массив указателей на четырехбайтный IP-адрес. Тем не менее, смещение первого указателя в массиве составляет 28 байт. Для ссылки на действительное расположение данных прибавьте к адресу структуры *HOSTENT* 28 байт и получите ссылку на четырехбайтную область с данными 0x9D36B9BA, представляющими собой IP-адрес 157.54.185.186. Затем можно взять 4 байта после записи со смещением 28 байт, получив в результате 0.

Если бы с этим именем компьютера было связано несколько IP-адресов, присутствовало бы и другое смещение, и потребовалось бы по аналогии с первым случаем изменить указатель. Точно такая же операция позволяет

исправить указатель и массив указателей, на который он ссылается. В данном примере у компьютера нет псевдонимов. Первая запись массива `__0`, и это означает, что в отношении данного поля какие-либо дополнительные действия не нужны. Последнее поле — `hjiame`, исправить которое достаточно просто. Следует лишь добавить смещение к адресу структуры `HOSTENT`, и поле будет указывать на начало оканчивающейся нулем строки.

Код, превращающий смещения в реальные адреса, прост, хотя и включает некоторые арифметические действия с указателями. Для корректировки поля `bname` подойдет следующая процедура настройки смещения.

```
hostent->h_name = (PCHAR)((DWORD_PTR)hostent->h_name) + (PCHAR)hostent,
```

Чтобы изменить массив указателей (например, поля `h_aliases` и `h_addrlist`), необходим более сложный код, который будет просматривать массив и изменять ссылки, пока не достигнет нулевой записи.

```

PCHAR «addr,                                     !
if (hostent->h_aliases)                           ;
<                                                 I
    addr = hostent->h_aliases = (PCHAR)((DWORD_PTR)hostent->h_aliases + ,
(PCHAR)hostent),
    while (addr)                                  л
    {
        addr = (PCHAR)((DWORD_PTR)addr + (PCHAR)0hostent),      !
        addr++;                                              ;

```

Этот код переходит от одной записи массива к другой и добавляет начальный адрес структуры `HOSTENT` к заданному смещению, в результате получается новое значение текущей записи. Конечно, по достижении нулевой записи просмотр массива прекращается. Данную операцию следует выполнить и для поля `b_addr_hst`. После того как смещения будут изменены, со структурой `HOSTENT` можно работать в обычном порядке.

## Резюме

Функции регистрации и разрешения имен могут показаться сложными, однако они обеспечивают значительную гибкость при разработке клиент-серверных приложений. Реальные ограничения на регистрацию имен связаны непосредственно с пространством имен. Удивительно, что при всей популярности пакета протоколов TCP/IP доступен единственный метод разрешения имен — DNS, который не обеспечивает требуемой гибкости. В доменных пространствах Windows 2000 и Windows NT доступен постоянный, не зависящий от протокола метод разрешения имен, обеспечивающий необходимую гибкость для разработки устойчивых приложений. Кроме того, приложения на основе протокола IPX/SPX доступны другие пространства имен (например, SAP), предоставляющие большинство из возможностей NTDS (за исключением независимости от протокола).



ления есть особый участник многоадресной группы — *корень* `c_root`. Остальные участники группы называются *листами* — `c_leaf`. В большинстве случаев корень (`c_root`) организует многоточечную группу, иницилируя соединения с любым количеством листьев (`c_leaf`). Иногда лист запрашивает членство в данной многоточечной группе позже. При этом для данной группы может существовать только один корневой узел. Пример корневой плоскости управления — протокол ATM.

Равноправная плоскость управления позволяет соединяться с группой любому участнику, то есть все участники группы являются листьями (узлами `c_leaf`). Каждый участник вправе присоединиться к многоточечной группе. Вы можете создать собственную схему членства в группе в рамках равноправной плоскости управления (тогда один из узлов станет корнем — `croot`), разработав собственный протокол членства. Впрочем, ваша схема группового членства будет по-прежнему основана на равноправной плоскости управления. Пример такой равноправной плоскости управления — многоадресная IP-рассылка.

На рис. 11-1 показаны различия между корневой и равноправной плоскостями управления. В корневой плоскости управления (слева) корень (`c_root`) должен явно опросить каждый лист (`c_leaf`) для присоединения к группе, в то время как в равноправной схеме (справа) к группе может присоединиться любой.

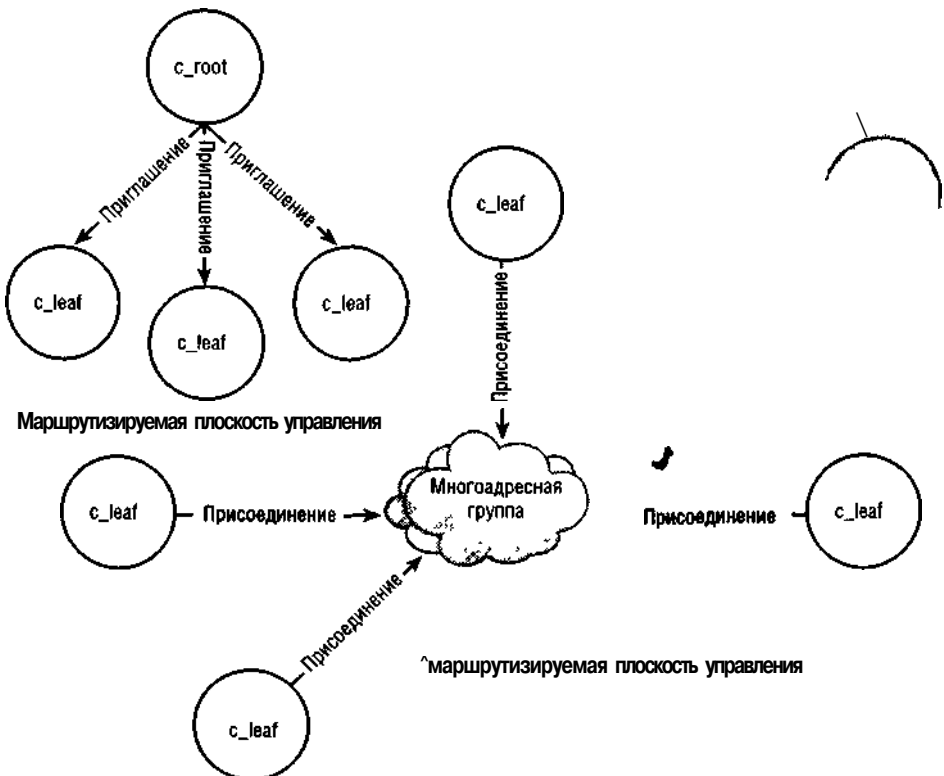


Рис. 11-1. Маршрутизируемые и ^маршрутизируемые плоскости управления

# Многоадресная рассылка

I T  
по!

*Многоадресная рассылка* (multicasting) — сравнительно новая технология, позволяющая отправлять данные от одного участника сети, а затем тиражировать множеству других, не создавая при этом большой нагрузки на сеть. Эта технология была разработана как альтернатива *широковещанию* (broadcasting), которое при активном использовании снижает пропускную способность сети. При многоадресной рассылке данные передаются в сеть, только если процессы, выполняемые на рабочих станциях в этой сети, запрашивают их. Не все протоколы поддерживают многоадресную рассылку. На платформах Win32 таких протоколов, доступных из Winsock, только два: IP и ATM.

Мы обсудим технологию многоадресной рассылки в целом, а также поясним, как она реализована в этих двух протоколах. Первоначально рассмотрим основы семантики многоточечных сетей, включая разные типы многоадресной рассылки, а так же основные характеристики IP и ATM. И наконец, опишем API-вызовы для многоадресной рассылки в Winsock 1 и Winsock 2. Ее поддержка для IP появилась в Winsock 1, а в Winsock 2 интерфейс многоадресной рассылки стал независимым от протокола.

Многоадресная рассылка поддерживается в Windows CE 2.1, 9x, NT 4 и 2000. Поддержка многоадресной рассылки по IP появилась в Windows CE лишь с версии 2.1. Все платформы, поддерживающие многоадресную рассылку, поддерживают ее по протоколу IP. Естественная поддержка ATM в Winsock появилась только в Windows 98 и 2000. Заметьте: это не мешает разрабатывать приложения многоадресной IP-рассылки для сетей ATM. Просто вы не сможете написать «родной» код многоадресной ATM-рассылки. Мы подробно рассмотрим этот вопрос в разделе, посвященном многоадресной рассылке в IP-сетях.

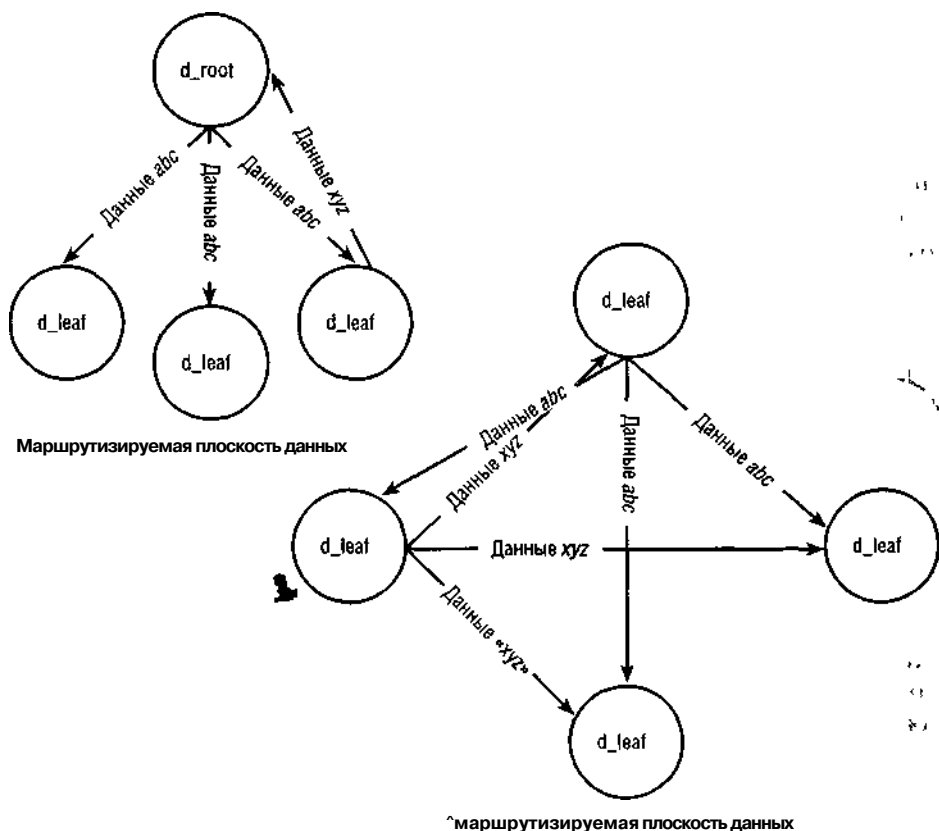
Есть еще одна проблема: некоторые устаревшие сетевые адаптеры не способны принимать и отправлять информацию по адресам многоадресной IP-рассылки. Большинство сетевых адаптеров произведенных за последние несколько лет поддерживают многоадресную IP-рассылку, но есть и исключения.

## Семантика многоадресной рассылки

Многоадресная рассылка обладает двумя важными характеристиками: *плоскостью управления* (control plane) и *плоскостью данных* (data plane). Первая определяет способ организации членства в группах, вторая — отражает способ распространения данных среди членов сети. Любая из этих плоскостей может быть корневой или равноправной. В корневой плоскости управ-

Плоскость данных также может быть маршрутизируемой или немаршрутизируемой. В маршрутизируемой плоскости данных есть участник, называемый *droot*. Передача данных происходит только между *d\_root* и всеми участниками многоточечного сетевого соединения, называемыми *dleaf*. Трафик может быть и одно-, и двунаправленным. Но маршрутизируемая плоскость данных подразумевает, что данные, передаваемые от одного *d\_leaf* будут приняты только *droot*, в то время как данные передаваемые от *d\_root* будут получены каждым *d\_leaf*.

Пример маршрутизируемой плоскости данных — АТМ. Рис. 11-2 показывает различия между маршрутизируемой или немаршрутизируемой плоскостью данных. В маршрутизируемой плоскости данных (слева) данные *abc* от *droot* передаются каждому *dleaf*. Данные *xyz*, передаваемые от *d\_leaf*, принимаются только *d\_root*. В этом отличие от немаршрутизированной плоскости (справа), где данные *abc* и *xyz* передаются каждому участнику сети вне зависимости от того, кто их отправил.



**Рис. 11-2. Маршрутизируемые и немаршрутизируемые плоскости данных**

Наконец, в немаршрутизируемой плоскости данных все члены группы могут отправлять данные всем остальным членам группы, и все получатели вправе отправлять данные в обратном направлении. Нет ограничений на то,

кто может получать или отправлять данные. Напомним, что многоадресная рассылка в сетях IP не маршрутизируется в плоскости данных.

Как видно, многоадресная рассылка в сетях ATM маршрутизируется и в плоскости управления, и в плоскости данных, в то время как многоадресная рассылка в сетях IP не маршрутизируема ни в одной из плоскостей. Помимо этих двух комбинаций могут существовать и другие. Например, совместно с маршрутизируемой плоскостью управления, где один узел решает, кто вправе присоединиться к группе, может существовать немаршрутизируемая плоскость данных, где данные, отправленные одним участником, будут доставлены остальными. Впрочем, ни один из поддерживаемых на сегодня протоколов в Winsock не ведет себя таким образом.

## Свойства многоадресной рассылки

В главе 5 мы обсудили, как перечислять записи протоколов и определять их свойства. Вся необходимая информация о протоколе также доступна в записи протокола в каталоге. Поле *divServiceFlags1* из структуры *WSAPROTOCOLINFO*, возвращаемой *WSAEnumProtocols*, содержит несколько интересующих нас бит. Если задан бит *XPI\_SUPPORT\_MULTIPOINT*, протокол поддерживает многоадресную рассылку. При заданном бите *XPI\_MULTIPOINT\_CONTROL\_PLANE* протокол поддерживает маршрутизируемую плоскость управления, иначе — нет. Заданный бит *XPI\_MULTIPOINT\_DATA\_PLANE* сообщает, что протокол поддерживает маршрутизируемую плоскость данных, если бит равен 0 — нет.

## Многоадресная рассылка в сетях IP

Многоадресная рассылка в сетях IP зависит от *групповых адресов* (multicast address). Например, если пять узлов хотят взаимодействовать путем многоадресной рассылки, они должны объединиться под неким групповым адресом. Затем любая информация, отправляемая одним узлом, будет передаваться каждому участнику группы, включая и тот узел, с которого была отправлена. Групповой адрес — IP-адрес класса D в диапазоне 224.0.0.0 — 239.255.255.255. Часть этих адресов зарезервирована для специальных целей, например, последние два — для использования протоколом IGMP, который мы вкратце рассмотрим.

Полный список зарезервированных адресов содержится в RFC 1700, где перечислены сетевые ресурсы особого назначения. Поддержкой этого списка занимается агентство IANA. Вот некоторые зарезервированные адреса:

И 224.0.0.0 — базовый адрес;

И 224.0.0.1 — все системы в данной подсети, \*

В 224.0.0.2 — все маршрутизаторы в данной подсети;

III 224.0.1.1 — протокол сетевого времени,

И 224.0.0.9 — групповой адрес RIP версии 2;

III 224.0.1.24 — групповой адрес WINS-сервера.

Во времена разработки протокола TCP/IP о многоадресной рассылке еще не думали, поэтому, чтобы протокол IP поддерживал ее, требовались коррек-

тивы. Мы уже рассказали о требовании IP, чтобы назначенные специальные адреса применялись только для многоадресного трафика. Кроме того, был разработан специальный протокол для управления клиентами многоадресной рассылки и их членством в группах. Представьте, что две рабочие станции в отдельных подсетях хотят присоединиться к одной многоадресной группе. Как это осуществить средствами IP? Вы не можете направлять информацию на групповой адрес наобум: сеть переполнится многоадресными данными. Для уведомления маршрутизаторов, что компьютер в сети хочет принимать данные, предназначенные группе, был разработан протокол IGMP.

## Протокол IGMP

Узлы многоадресной рассылки используют IGMP для сообщения маршрутизаторам, что компьютер в маршрутизируемой подсети хочет присоединиться к определенной многоадресной группе. IGMP — основа реализации многоадресной рассылки в IP, чтобы рассылка шла корректно, его должны поддерживать все маршрутизаторы между двумя узлами. Например, если компьютеры А и В объединены в группу 224.1.2.3 и между ними три маршрутизатора, то все эти маршрутизаторы должны быть совместимы с IGMP. Любой другой маршрутизатор просто удалит полученные данные многоадресной рассылки. Когда приложение присоединяется к многоадресной группе, на адрес всех маршрутизаторов в подсети (224.0.0.2) отправляется IGMP-команда join. Эта команда уведомляет маршрутизатор о наличии клиентов, заинтересованных в получении данных с определенного группового адреса. Когда маршрутизатор получит данные, предназначенные для этого адреса, он переправит их в подсеть заинтересованного клиента.

Присоединяясь к группе, конечная точка указывает параметр *время жизни* (time-to-live, TTL) — сколько маршрутизаторов может пересекать приложение на данной конечной точке при отправке и получении данных.

Например, если вы пишете приложение многоадресной рассылки, которое присоединяется к группе X с параметром TTL, равным 2, то группе маршрутизаторов в локальной подсети будет отправлена соответствующая команда join. Маршрутизаторы этой подсети примут команду и начнут пересылать вещательные данные, предназначенные для указанного адреса. Затем каждый маршрутизатор уменьшит TTL на 1 и передаст команду присоединения в соседние сети. Маршрутизаторы этих сетей выполняют ту же операцию, снова уменьшая TTL. В результате TTL станет равным 0, и команда дальше передаваться не будет. Так TTL ограничивает зону передачи вещательных данных.

Зарегистрировав одну или несколько групп многоадресной рассылки, маршрутизатор начнет периодически отправлять сообщение группового опроса всем узлам (224.0.0.1) для каждого группового адреса, который был зарегистрирован командой join. Если клиенты этой сети все еще используют групповой адрес, они ответят IGMP-сообщением, чтобы маршрутизатор продолжал перенаправлять данные, связанные с этим адресом. Даже если клиент явно покидает многоадресную группу, используя любой из соответствующих методов Winsock, фильтр на маршрутизаторе сбрасывается не сразу.

К сожалению, это может привести к нежелательным последствиям: клиент выйдет из группы А и немедленно присоединится к группе В. Пока маршрутизатор не произведет опрос и не получит отрицательный ответ, он будет пересылать в сеть информацию, предназначенную для обеих групп. Это особенно плохо, если суммарный размер передаваемой информации больше пропускной способности сети. Вот здесь и проявляется преимущество IGMP, версии 2: она позволяет клиенту явно отправить маршрутизатору сообщение *leave* о выходе из группы, чтобы немедленно остановить передачу данных для этого адреса. Конечно, маршрутизатор поддерживает контрольный счетчик количества клиентов для каждого адреса, поэтому пока все клиенты подсети не откажутся от адреса, данные, предназначенные для него, будут распространяться.

Windows 98 и Windows 2000 изначально поддерживают IGMP версии 2. Последняя версия обновления Windows 95 содержит IGMP 2. Для Windows NT 4 поддержка IGMP 2 включена в Service Pack 4. Предыдущие версии пакетов обновлений и базовая версия ОС поддерживают только IGMP 1. Подробную спецификацию IGMP 1 и 2 см. в RFC 1112 и RFC 2236, соответственно.

## Листовые узлы IP

Процесс присоединения группы многоадресной рассылки в сетях IP прост, так как каждый узел является листовым (*leaf*) и поэтому выполняет одинаковые шаги для присоединения к группе. Поскольку многоадресная рассылка в сетях IP доступна в Winsock 1 и 2, есть два механизма API-вызовов для выполнения одной и той же операции. Вот основные шаги для выполнения многоадресной рассылки средствами Winsock 1.

1. С помощью функции *socket* создайте сокет семейства адресов *AF\_INET* типа *SOCK\_DGRAM*. Никаких особых флагов, указывающих на многоадресный характер сокета, не требуется, поскольку функция *socket* не имеет флаговых параметров.
2. Свяжите сокет с локальным портом, если хотите получать данные от группы.
3. Вызовите функцию *setsockopt* с параметром *IP\_ADD\_MEMBERSHIP* и указанием адресной структуры для группы, к которой хотите присоединиться.

Если вы используете Winsock 2, то шаги 1 и 2 — те же, а в шаге 3 вызовите функцию *WSAJoinLeaf* для добавления своего адреса в группу. Эти две функции и их различия подробно обсуждаются далее, в разделах, посвященных многоадресной рассылке средствами Winsock.

Напомним: приложению не нужно присоединяться к группе, если оно только отправляет данные. При отправке данных в многоадресную группу применяется обычный UDP-пакет, только направляется он по групповому адресу. Если вам нужно принимать рассылку, тогда присоединитесь к группе. За исключением требования членства в группе, IP-рассылка характеризуется так же, как обычный протокол UDP: не требует логического соединения, не надежна и т. д.

## Реализация IP-рассылки

IP-рассылка поддерживается всеми платформами Windows (кроме Windows CE до версии 2.1), но реализуется немного по-разному на каждой. Мы уже упоминали, что сетевой адаптер должен поддерживать многоадресную рассылку — уметь аппаратно добавлять многоадресные фильтры в интерфейс. Групповые IP-адреса используют специальный MAC-адрес, который содержит зашифрованный IP-адрес. Поэтому сетевые адаптеры могут легко выяснить, является ли входящий пакет рассылкой и, изучив MAC-заголовок, понять, по какому групповому IP-адресу он отправлен. Механизм шифрования описан в RFC 1700. Мы не будем обсуждать его здесь, потому что он не имеет отношения к Winsock.

Впрочем, Windows 95 и Windows NT 4 выполняют многоадресную рассылку за счет перевода сетевого адаптера в смешанный режим. Тогда сетевой адаптер извлекает все принимаемые пакеты, и сетевой драйвер изучает MAC-заголовки в поисках многоадресных данных, предназначенных группе, членом которой является процесс, выполняющийся на рабочей станции. Так как эта фильтрация осуществляется программно, она не столь оптимальна, как аппаратная.

Windows 98 и Windows 2000 ведут себя иначе, поскольку знают о возможности адаптеров добавлять аппаратные фильтры. Большинство сетевых адаптеров поддерживают 16 или 32 аппаратных фильтра. Единственное ограничение Windows 98: как только будут заняты все аппаратные фильтры, процесс на компьютере не сможет присоединиться к группе. Если аппаратный предел превышен, то операция присоединения завершится с ошибкой *WSAENOBUFFS*. Windows 2000 более устойчива: когда все аппаратные фильтры израсходованы, она переводит сетевой адаптер в смешанный режим и функционирует аналогично Windows 95 и Windows NT 4.

## Многоадресная рассылка в сетях ATM

«Родной» ATM через Winsock тоже поддерживает многоадресную рассылку, предлагая куда большие возможности по сравнению с IP. Помните, что ATM поддерживает маршрутизируемые плоскости управления и данных. То есть когда многоадресный сервер (или с\_root) существует, он контролирует состав групп, а также маршруты передачи внутри группы.

Важно также, что в ATM-сетях IP-трафик может передаваться поверх ATM. Такая конфигурация позволяет ATM-сетям эмулировать IP-сеть путем сопоставления IP-адресов собственным ATM-адресам. IP поверх ATM позволяет выбрать: применять ли многоадресную рассылку в сетях IP, которая будет транслироваться на уровень ATM, или использовать собственные возможности ATM для рассылки. Поведение IP-рассылки в ATM при использовании IP поверх ATM точно такое же, как в обычной IP-сети. Единственное исключение: отсутствие IGMP, поскольку все многоадресные вызовы преобразуются в «родные» команды ATM.

Кроме того, можно сконфигурировать в ATM-сети одну или несколько эмулирующих локальных сетей (LANE). Цель LANE — сделать так, чтобы ATM-

сеть выглядела как «обычная», где можно использовать разные протоколы: IPX/SPX, NetBEUI, TCP/IP, IGMP, ICMP и др. Тогда рассылка в IP-сети фактически выглядит как многоадресная рассылка в сети Ethernet, а значит, доступен и протокол IGMP.

Мы упоминали, что ATM поддерживает маршрутизируемые плоскости управления и данных: когда вы создаете многоадресную группу, образуется корневой узел, который приглашает листовые. В Windows 2000 сейчас поддерживается только соединение, инициированное корнем, то есть лист не может попросить о включении в группу. Корневой узел (в качестве маршрутизируемой плоскости данных) отправляет данные только в одном направлении: от корня к листьям.

Еще одно заметное отличие от IP — ATM не нуждается в особых адресах. Необходимо только, чтобы корень знал адреса всех листьев, которые он будет приглашать. При этом только один корневой узел может присутствовать в многоадресной группе. Как только другая конечная точка ATM-сети начинает приглашать другие листья, это объединение отделяется от первой группы.

## Листовые узлы ATM

Создать листовой узел в многоадресной группе просто. В ATM-сети лист должен слушать приглашение от корня. Вот что необходимо сделать.

1. Используя функцию *WSASocket*, создайте сокет семейства адресов *AF\_ATM* с флагами *WSAJMGT^ULTIPOimjO\_LEAFviWSAJJAG\_MULTIPOINT\_D\_LEAF*.
2. Свяжите сокет с локальным ATM-адресом и портом функцией *bind*.
3. Вызовите функцию *listen*.
4. Дождитесь приглашения, используя функции *accept* или *WSAAccept* в зависимости от применяемой модели ввода-вывода. (Более полное описание моделей ввода-вывода Winsock см. в главе 8.)

Установив соединение, листовой узел может получать данные от корня. Помните, что при рассылке в ATM данные передаются в одном направлении: от корня к листьям.

**ПРИМЕЧАНИЕ** В настоящее время Windows 98 и Windows 2000 способны поддерживать только один листовой узел ATM одновременного?, то есть только один процесс во всей системе может быть листовым *nl* участником любого многоадресного сеанса ATM.

## Корневые узлы ATM

101

Создать корневой узел еще легче, чем лист ATM.

1. С помощью функции *WSASocket* создайте сокет адресного семейства *AF\_ATM* с флагами *WSA\_FLAG\_MULTIPOINT\_C\_ROOT* и *WSA\_FLAG\_JITMULTIPOINTJJDZOOT*.
2. Вызовите функцию *WSAJoinLeafn* передайте ей в качестве параметра ATM-адрес каждой конечной точки, которую вы хотите пригласить.



Корневой узел может пригласить любое число конечных точек, но для каждой нужен отдельный вызов *WSAJoinLeaf*.

## Многоадресная рассылка с использованием Winsock

Теперь рассмотрим соответствующие API-вызовы Winsock. Есть два метода организации многоадресной рассылки в IP-сетях, в зависимости от применяемой версии Winsock. Первый метод, доступный в Winsock 1, заключается в использовании параметров сокета для присоединения к группе. Winsock 2 вводит новую функцию присоединения — *WSAJoinLeaf*, которая не зависит от базового протокола.

Метод с использованием Winsock 1 мы обсудим первым. Он широко распространен, главным образом, потому что унаследован от сокетов Беркли.

### Рассылка средствами Winsock 1

В Winsock 1 для присоединения и выхода из многоадресной группы применяется функция *setsockopt* с параметрами *IP\_ADD\_MEMBERSHIP* и *IP\_DROP\_MEMBERSHIP*. При использовании любого из параметров вы должны передать структуру *ipjnreq*.

```
struct ipjnreq
{
    struct in_addr  imrjnultiaddr;
    struct in_addr  imr_interface;
};
```

Поле *imrjnultiaddr* задает группу, к которой нужно присоединиться, а *imrjnterface* — локальный интерфейс для отправки данных. Если указать *INADDR\_ANY* для *imrjnterface*, будет использован интерфейс по умолчанию. Либо если IP-адресов несколько, задайте адрес локального интерфейса явно. Следующий пример иллюстрирует присоединение к группе 234.5.6.7.

```
SOCKET      s;
struct ipjnreq  ipmr;
SOCKADDR_IN  local;
int          len = sizeof(ipmr);

s = socket(AF_INET, SOCK_DGRAM, 0);

local.sin_family = AF_INET;
local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_port = htons(10000);

ipmr.imrjnultiaddr.s_addr = inet_addr("234.5.6.7");
ipmr.imr_interface.sjiddr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&local, sizeof(local));
```

```
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP,
(char O&ipmr, &len);
```

Для выхода из группы достаточно вызвать функцию *setsockopt* с параметром *IP\_DROP\_MEMBERSHIP*, передав ей структуру *ipjnreq* с теми же значениями, которые использовались для присоединения к группе:

```
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP,
(char O&ipmr, &len);
```

### Пример многоадресной рассылки в IP-сети средствами Winsock 1

В листинге 11-1 приведен пример программы многоадресной рассылки, которая присоединяется к заданной группе и затем действует как отправитель или получатель (в зависимости от аргументов командной строки). В этом примере отправитель только отправляет, а получатель просто ждет в цикле входящие данные. Конечно, возможна ситуация, выходящая за рамки примера: если отправитель отправляет данные группе и также слушает данные, будучи в составе этой группы, то отправленные данные вернуться к нему во входную очередь. Такой процесс называется петлей и происходит, только когда вы ведете рассылку в IP-сети. Далее мы обсудим, как этого избежать.

#### Листинг 11-1. Пример организации многоадресной рассылки средствами Winsock 1

```
II Название модуля: Mcaetwsi.c

<include <windows.h>
<include <winsock.h>

<include <stdio.h>
<include <stdlib.h>

#define MCASTADDR "234.5.6.7"
#define MCASTPORT 25000
#define BUFSIZE 1024
#define DEFAULT COUNT 500

BOOL bSender = FALSE, // Действовать как отправитель?
bLoopBack = FALSE; // Запретить образование петли?

DWORD dwInterface, // Локальный интерфейс для привязки
dwMulticastGroup, // Многоадресная группа, к которой присоединиться'
dwCount; // Количество сообщений для отправки/приема

short iPort; // Номер используемого порта

II
II Функция: usage
```

см. след. стр.

## Листинг 11-1. (продолжение)

```

* д      j^p"
ц      • i      .< * i^rto)
// Описание:
// Вывод информации об использовании и выход

```

```

void usage(char *programe)

printf("usage: %s -s -nrstr -p.int -i:str -l -n:int\n",
      programe);
printf(" -s      Act as server (send data); otherwise\n");
printf("          receive data.\n");
printf(" -m:str Dotted decimal multicast IP address to join\n");
printf("          The default group is. Xs\n", MCASTADDR);
printf(" -p.int Port number to use\n"),
printf("          The default port is: Xd\n", MCASTPORT);
printf(" -i:str Local interface to bind to; by default \n");
printf("          use INADDR_ANY\n");
printf(" -l      Disable loopback\n");
printf(" -n:int Number of messages to send/receive\n");
ExitProcess(-i);

```

```

// функция: ValidateArgs

```

```

// Описание:
// Анализ параметров командной строки и установка некоторых глобальных
// флагов в зависимости от значений

```

```

void ValidateArgs(int argc, char **argv)

int      i;

dwInterface = INADDR_ANY;
dwMulticastGroup = inet_addr(MCASTADDR);
lPort = MCASTPORT;
dwCount = DEFAULT_COUNT;

for(i = 1; i < argc; i++)

if ((argv[i][0] == '-' || (argv[i][0] == V))

switch (tolower(argv[i][1]))

case 's' // Отправитель
    bSender = TRUE;
    break;
case 'm'. // Многоадресная группа
    if (strlen(argv[i]) > 3)
        dwMulticastGroup = inet_addr(&argv[i][3]);

```

**Листинг 11-1. (продолжение)**

```

break;
case 'i': // Локальный интерфейс
    if (strlen(argv[i]) > 3)
        dwInterface = inet_addr(&argv[i][3]);
    break;
case 'p': // Номер порта
    if (strlen(argv[i]) > 3)
        IPort = atoi(&argv[i][3]);
    break;
case '1': // Запретить образование петли?
    bLoopBack = TRUE;
    break;
case 'n': // Количество сообщений для отгцмМИЦ/^рЯМЯГтя^
    dwCount = atoi(&argv[i][3]);
    break;
default:
    usage(argv[0]);
    break;
}
return;

//
// Функция: main
//
// Описание:
// Анализ параметров командной строки, загрузка библиотеки Wmsock
// создание сокета и присоединение к многоадресной группе. Если програти*
// запущена как отправитель, то начинается отправка сообщений
// группе, иначе вызывается recvfrom() для
// чтения сообщений, отправленных группе.
//
im: main(int argc, char **argv)
{
    WSADATA wsd;
    struct sockaddr_in local,
    remote,
    from;
    struct ipjnreq mcast;
    SOCKET sockM;
    TCHAR recvbuf[BUFSIZE],
    sendbuf[BUFSIZE];
    int len = sizeof(struct sockaddr_in),
    optval,
    ret;

```

см. след. апр.

## Листинг 11-1. (продолжение)

```

DWORD          i=0;

// Анализ командной строки и загрузка Winsock

ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)

    printf("WSAStartup failed\n");
    return -1;

// Создание сокета. В Winsock 1 вам не потребуется никаких специальных
// флагов для указания многоадресной рассылки

if ((sockM = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)

    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;

// Привязка сокета к локальному интерфейсу. Это нужно для приема данных {

local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sockM, (struct sockaddr *)&local,
    sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}

// Настройка структуры im_req для указания группы, к которой мы
// хотим присоединиться, и интерфейса
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;

mcast.imr_multiaddr.s_addr = dwMulticastGroup;
mcast.imr_interface.s_addr = dwInterface;

if (setsockopt(sockM, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)

    printf("setsockopt(IP_ADD_MEMBERSHIP) failed: %d\n",

```

Чистинг 11-1. (продолжение')

```

        WSAGetLastErrorO);
        closesocket(sockM);
        WSACleanupO;
        return -1;

// Настройка значения TTL (по умолчанию - 1)

optval = 8;
if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_TTL,
    (char O&optval, sizeof(int)) == SOCKET.ERROR)

    printf("setsockopt(IP_MULTICAST_TTL) failed: Xd\n",
        WSAGetLastErrorO);
    closesocket(sockM);
    WSACleanupO;
    return -1;

// Запрет петли, если это было выбрано. Заметьте,
// что в Windows NT 4 и Windows 95 петлю запретить нельзя

if (bLoopBack)

    optval = 0;
    if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)

        printf("setsockopt(IP_MULTICAST_LOOP) failed: Xd\n",
            WSAGetLastErrorO);
        closesocket(sockM);
        WSACleanupO;
        return -1;
    >

if (lbSender)                // Клиент

    // Прием порции данных

    for(i = 0; i < dwCount;

        if ((ret = recvfrom(sockM, recvbuf, BUFSIZE, 0,
            (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
        {
            printf("recvfrom failed with: Xd\n",
                WSAGetLastErrorO);
            closesocket(sockM);
            WSACleanupO;

```

см. след. стр.

Листинг 11-1.    (продолжение)

```

        return -1;
    }
    recvbuf[ret] = 0;
    printf("RECV:  `Xs' from <Xs>\n",  recvbuf,
        inet_ntoa(from.sin_addr));
}
else
    // Сервер

// Отправка порции данных
//
for(i = 0; i < dwCount; i++)
    sprintf(sendbuf, "server 1: This is a test: Xd", i);
    if (sendto(sockM, (char *)sendbuf, strlen(sendbuf), 0,
        (struct sockaddr *)&remote,
        sizeof(remote)) == SOCKET_ERROR)
    {
        printf("sendto failed with: Xd\n"ipbncW # f> ГО внобоШ а OTS< \\
            WSAGetLastErrorO);
        closesocket(sockM);
        WSACleanupO;
        return -1;
    }
    Sleep(500);

}
// Выход из группы

if (setsockopt(sockM, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_DROP_MEMBERSHIP) failed: Xd\n",
        WSAGetLastErrorO);
    closesocket(sockM);
    WSACleanupO;
    return 0;
}

```

Одно предостережение при организации рассылки в Winsock 1: используйте корректные заголовочный файл и библиотеки для компоновки. Если вы загружаете библиотеку Winsock 1.1, подключите Winsock.h и компоунуйте его с Wsock32.lib. Для версии 2 или выше подключайте Winsock2.h и Ws2tcpip.h и компоунуйте их с Ws2\_32.lib. Это необходимо, поскольку существует два множества значений для констант *IP\_ADD\_MEMBERSHIP*, *IP\_DROP\_MEMBERSHIP*, *IPJWULTICASTJF* и *IP ^MULTICAST LOOP*. Исходная спецификация значений, написанная Стивеном Дирингом (Stephen Deering), никогда офи-

циально не включалась в спецификацию Winsock. В результате они изменились в спецификации Winsock 2. Конечно, если вы используете раннюю версию Winsock, компоновка с `wsock32.lib` решит все проблемы и константы получат корректные значения даже при запуске на компьютере с Winsock 2.

## Рассылка средствами Winsock 2

Многоадресная рассылка в Winsock 2 немного сложнее, чем в Winsock 1, но поддерживает разные протоколы, что дает дополнительные возможности, например, использовать Quality of Service (QoS). Кроме того, рассылка в Winsock 2 позволяет применять протоколы, поддерживающие маршрутизируемые схемы. Параметры сокета больше не нужны для инициализации членства в группе — им на смену пришла функция *WSAJoinLeaf*.

```
SOCKET WSAJoinLeaf(  
    SOCKET s,  
    const struct sockaddr FAR * name,  
    int namelen,  
    LPWSABUF lpCallerData,  
    LPWSABUF lpCalleeData,  
    LPQOS lpSQOS,  
    LPQOS lpGQOS,  
    DWORD dwFlags  
);
```

Параметр 5 — описатель сокета, возвращенный *WSASocket*. Переданный сокет должен быть создан с соответствующими флагами, иначе *WSAJoinLeaf* вернет ошибку *WSAEINVAL*. Помните о необходимости задать два флага: один показывает, будет ли этот сокет маршрутизируемым в плоскости управления, другой — будет ли сокет маршрутизируемым в плоскости данных. Флаги для плоскости управления — *WSA\_FLAG\_MULTIPOINT\_C\_ROOT* и *WSA\_FLAG\_MULTIPOINT\_C\_LEAF*. Флаги для плоскости данных — *WSA\_FLAG\_MULTIPOINT\_D\_ROOT* и *WSA\_FLAG\_MULTIPOINT\_D\_LEAF*.

Второй параметр — структура *SOCKADDR*, специфичная для используемого протокола. Для маршрутизируемых схем управления (например, ATM) этот адрес указывает клиента, которого нужно пригласить, а для немаршрутизируемых схем (например, IP) — это адрес группы, к которой узел присоединяется.

Параметр *namelen* — длина в байтах параметра *name*. Параметр *lpCallerData* используется для передачи буфера данных партнеру в установленном сеансе, а *lpCalleeData* указывает буфер, который будет передан обратно. Эти два параметра пока не реализованы на платформах Windows и должны быть равны *NULL*. Параметр *lpSQOS* задает структуру *FLOWSPEC*, указывающую требуемую пропускную способность для приложения (подробнее о QoS — в главе 12.) Параметр *lpGQOS* игнорируется, поскольку ни одна из платформ Windows не поддерживает группы сокетов. Последний параметр — *dwFlags*, показывает роль узла: отправка данных, прием данных, или и то, и другое. Возможные значения — *JL\_SENDER\_ONLY*, *JL\_RECEIVER\_ONLY*, *JL\_BOTH*.



Функция возвращает описатель *SOCKET* для сокета, связанного с многоадресной группой. Если вызов *WSAJoinLeaf* выполнен с асинхронным (неблокирующим) сокетом, возвращаемый описатель сокета не пригоден к использованию, пока не завершится операция присоединения. Например, в этом случае после вызова *WSAAsyncSelect* или *WSAEventSelect* описатель не будет действителен, пока исходный сокет 5 не вернет сообщение *FD\_CONNECT*. Сообщение *FD\_CONNECT* генерируется только в маршрутизируемой схеме управления, в которой параметр *name* задает конкретный адрес конечной точки.

В табл. 11-1 перечислены обстоятельства, при которых приложение получает сообщение *FD\_CONNECT*. Вы вправе аннулировать ожидающий выполнения запрос на присоединение для этих неблокирующих режимов, вызвав *closesocket* на исходном сокете. Корневой узел в многоточечном сеансе может вызывать *WSAJoinLeaf* один или несколько раз, чтобы добавить несколько листовых узлов; впрочем, в каждый момент времени ожидать выполнения может только один запрос на многоточечное соединение.

Табл. 11-1. Действия *WSAJoinLeaf*

Плоскость управления	s	Имя	Действие	Прием уведомления <i>FDCONNECT</i>	Возвращение описателя сокета
Маршрутизируемая	c_root	Адрес листа	Корень приглашает лист	Да	Используется для уведомления <i>FDCLOSE</i> и для отправки данных только этому листу
	c_jeaf	Адрес корня	Лист инициирует соединение с корнем	Да	Дубликат* .тэнсп- 'ыесж..-
Немаршрутизируемая	c_root	—	Невозможная комбинация	—	— оэроп: АЮЧ\TV
	c_jeaf	Адрес группы	Лист присоединяется к группе	Нет	Дубликат 5 >ОШ <\

Как мы уже упоминали, запрос на присоединение для неблокирующих сокетов не может завершиться немедленно. Если сокет переведен в неблокирующий режим посредством *ioctlsocket* и команды *FIONBIO*, вызов *WSAJoinLeaf* не вернет ошибку *WSAEWOULDBLOCK*, поскольку эта функция фактически выдаст сообщение об успешном запуске. Заметьте, что в асинхронной модели ввода-вывода единственный способ узнать об успешном запуске — сообщение *FD\_CONNECT*. (Подробнее об асинхронных моделях *WSAAsyncSelect* и *WSAEventSelect* — в главе 8.) Блокирующие сокеты не могут уведомить приложение об удачном или ошибочном завершении *WSAJoinLeaf*. Другими словами, применять неблокирующие сокеты не стоит, поскольку нет однозначного способа определить успешно ли присоединение к группе, пока сокет не будет задействован в последующих вызовах Winsock (которые вернут ошибку, если присоединения не произошло).

Описатель сокета, возвращенный *WSAJoinLeaf*, зависит от того, является ли сокет маршрутизируемым или листовым узлом. Для маршрутизируемого узла параметр *name* указывает адрес конкретного листа, который приглашается в многоточечный сеанс. Чтобы *c\_root* поддерживал членство листьев, *WSAJoinLeaf* возвращает для листа новый описатель сокета.

Новый сокет имеет те же свойства, что и корневой описатель, использованный для приглашения, включая любые асинхронные события, зарегистрированные по асинхронным моделям ввода-вывода типа *WSAEventSelect* и *WSAAsyncSelect*. Впрочем, эти новые сокеты следует применять только для получения уведомления *FD\_CLOSE* от листа. Любые данные, которые нужно отправить многоадресным группам, должны отправляться через сокет *c\_root*. Иногда вы можете отправить данные на сокет, возвращенный *WSAJoinLeaf*, но их получит только лист, соответствующий этому сокету (это позволяет протокол ATM). Наконец, чтобы удалить листовую узел из многоточечного сеанса, корень просто вызывает *closesocket* на сокете, соответствующем этому листу.

С другой стороны, когда *WSAJoinLeaf* вызывается с листовым узлом, параметр *name* задает адрес либо корневого узла, либо многоадресной группы. В первом случае присоединение иницируется листом, что в настоящее время не поддерживает ни один протокол (спецификация ATM UNI 4.0 будет делать это). Второй пример — рассылка в IP. В любом случае описатель сокета, возвращенный *WSAJoinLeaf* — это тот же описатель сокета, что был передан в 5. Когда вызов *WSAJoinLeaf* служит для присоединения со стороны листа, корневой узел слушает входящие соединения, используя методы *bind*, *listen* и *accept/WSAAccept*, как обычно на сервере. Когда приложение хочет удалить себя из многоадресного сеанса, вызывается *closesocket* на том сокете, который прекращает членство (при этом также освобождаются ресурсы сокета). Табл. 11-1 резюмирует действия, выполняемые в зависимости от типа плоскости управления, и параметры, которые передаются на сокет. С ее помощью также можно определить, возвращается ли новый описатель сокета после удачного вызова функции и получает ли приложение уведомление *FD\_CONNECT*.

Допустим, приложение вызывает *accept* или *WSAAccept*, чтобы ожидать приглашения от корня, либо выполняет роль корня, чтобы ожидать запросы на присоединение от листьев. Тогда функция возвращает сокет, который является описателем сокета *c\_leaf* (такой же возвращает *WSAJoinLeaf*). Для работы с протоколами, которые поддерживают соединения, инициированные как корнем, так и листом, в качестве входного параметра для *WSAJoinLeaf* Допустимо передавать слушающий сокет *c\_root*.

После вызова *WSAJoinLeaf* возвращается новый описатель сокета. Этот описатель не применяется для отправки и приема данных: он просто показывает, что приложение — член многоадресной группы. Для операций отправки и приема используется исходный описатель сокета, полученный от *WSASocket* и затем переданный в *WSAJoinLeaf*. Вызов *closesocket* для нового описателя прекратит членство приложения в группе. В результате вызова *closesocket* на сокете *c\_root* все связанные с ним узлы *c\_leaf*, использующие асинхронную модель ввода-вывода, получают уведомление *FD\_CLOSE*.

## Пример многоадресной рассылки в IP-сети средствами Winsock 2

Листинг 11-2 содержит программу Mcastws2.c, которая иллюстрирует присоединение и выход из многоадресной группы с помощью *WSAJoinLeaf*. Это измененный пример организации IP-рассылки средствами Winsock 1. Отличие в том, что вызовы присоединения (выхода) здесь переписаны под *WSAJoinLeaf*.

### Листинг 11-2. Пример организации многоадресной рассылки средствами Winsock 2

```
// Модуль: Mcastws2.c
//
#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR    "234.5.6.7"
#define MCASTPORT    25000
#define BUFSIZE      1024
#define DEFAULT_COUNT 500

BOOL bSender = FALSE,    // Действовать как отправитель?
     bLoopBack = FALSE;  // Запретить образование петли?

DWORD dwInterface,       // Локальный интерфейс для привязки
     dwMulticastGroup,    // Многоадресная группа, к которой присоединиться
     dwCount;             // Количество сообщений для отправки/приема

short iPort;              // Номер используемого порта

//
// Функция: usage
//
// Описание:
// Вывод информации об использовании и выход
//
void usage(char *programe)
{
    printf("usage: Xs -s -m:str -p:int -i:str -l -n:int\n",
           programe);
    printfC    -s      Act as server (send data); otherwise\n");
    printfC    receive data.\n");
    printfC    -m:str   Dotted decimal multicast IP address "
               "to Join\n");
    printfC    The default group is: Xs\n", MCASTADDR);
    printfC    -p:int   Port number to use\n");
    printfC    The default port is: Xd\ri", MCASTPORT);
    printf("    -i:str   Local interface to bind to; by default \n");
```

```
printfC          use INADDR_ANY\n");
printf("  -1      Disable loopback\n");
printfC  -n:int  Number of messages to send/receive\n"); »b
ExitProcess(-i);      »!l
```

И Функция: `ValidateArgs`

```
// Анализ параметров командной строки и уопммк* некоторых глобальных
// флагов в зависимости от значений
```

```
//                                     1,                               nie» -««дону* \\
void ValidateArgs(int argc, char **argv)                                //
```

[illegible]

```
for(i=1; i < argc ;:
```

```

{
if ((argv[i][0] == '-') || (argv[i][0]
{
switch (tolower(argv[i][1]))
{
case 's': //Отправитель «Те-
bSender = TRUE;
break;

case 'пГ: // Многоадресная группа
if (strlen(argv[i]) > 3)
dwMulticastGroup = inet_addr(4argv[i][3]);
break;

case 'i': // Локальный интерфейс
if (strlen(argv[i]) > 3)
dwInterface = met_addr(4argv[i][3]);
break;

case 'p': // Номер порта
if (strlen(argv[i]) > 3)
iPort = atoi(&argv[i][3]);
break;

case '1': // Запретить образование петли?
bLoopBack = TRUE;
break;

case 'n': // Количество сообщений для отправки/приема

```

*см. след. стр.*

Листинг 11-2.    (продолжение)

Г Ф»

```

    dwCount = atoi(&argv[i][3]);
    break;
default:
    usage(argv[0]);
    break;

return;

// Функция: main

// Описание:
//   Анализ параметров командной строки, загрузка библиотеки Winsock,
//   создание сокета и присоединение многоадресной группе. Если программа
//   запущена как отправитель, то начинается отправка сообщений
//   группе, иначе вызывается recvfrom для
//   чтения сообщений, отправленных группе.
//
int main(int argc, char **argv)
{
    WSADATA          wsdata;
    struct sockaddr_in local,
                    remote,
                    from;
    SOCKET           sock, sockM;
    TCHAR            recvbuf[BUFSIZE],
                    sendbuf[BUFSIZE];
    int              len = sizeof(struct sockaddr_in),
                    optval,
                    ret;
    DWORD            i=0;

    // Анализ командной строки и загрузка Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsdata) != 0)
    {
        printf("WSAStartup() failed\n");
        return -1;
    }

    // Создание сокета. В Winsock 2 нужно задать многоадресные
    // свойства, с которыми будет использоваться сокет
    //
    if ((sock = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0,
        WSA_FLAG_MULTIPOINT_C_LEAF
```

**Листинг 11-2.** *(продолжение)*

```

| WSA_FLAG_MULTICAST |
| WSA_FLAG_OVERLAPPED) == INVALID_SOCKET)

printf("socket failed with: %d\n", WSAGetLastError());
WSACleanup();
return -1;

}

// Привязка сокета к локальному интерфейсу. Это нужно для приема сообщений
local.sin_family = AF_INET;
local.sin_port = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sock, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}

// Настройка структуры SOCKADDR_IN, описывающей многоадресную
// группу, к которой мы хотим присоединиться
remote.sin_family = AF_INET;
remote.sin_port = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;
//
// Настройка значения TTL

optval = 8;
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
        (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
        WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}

// Запрет петли, если это было выбрано
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
    {

```

см. след. стр.

## Листинг 11-2. (продолжение)

```

printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n", WSAGetLastErrorO);
WSAGetLastErrorO);
closesocket(sock);
WSACleanupO;
return -1;
}

// Присоединение к многоадресной группе. Заметьте: sockM
// не используется для отправки или получения данных. Он использует
// когда мы хотим выйти из многоадресной группы.
// Вы просто вызываете на нем closesocket().

if ((sockM = WSAJoinLeaf(sock, (SOCKADDR *)&remote,
    sizeof(remote), NULL, NULL, NULL,
    JL_BOTH)) == INVALID_SOCKET)

    printf("WSAJoinLeaf() failed: %d\n", WSAGetLastErrorO);
    closesocket(sock);
    WSACleanupO;
    return -1;

if (IbSender) // Получатель

    // Прием порции данных

    for(i = 0; i < dwCount; i++)
    {
        if ((ret = recvfrom(sock, recvbuf, BUFSIZE, 0,
            (struct sockaddr *)&from, ilen)) == SOCKET_ERROR)
        {
            printf("recvfrom failed with: %d\n", WSAGetLastErrorO);
            closesocket(sockM);
            closesocket(sock);
            WSACleanupO;
            return -1;
        }
        recvbuf[ret] = 0;
        printf("RCV: '%s' from %s\n", recvbuf, inet_ntoa(from.sin_addr));
    }

else // Отправитель

    // Отправка порции данных

    for(i = 0; i < dwCount;

```

Листинг 11-2. (продолжение)

.8-IT

```

sprintf(sendbuf, "server 1: This is a test: Xd", i);
if (sendto(sock, (char *)sendbuf, strlen(sendbuf), 0,
    (struct sockaddr *)&remote,
    sizeof(remote)) == SOCKET_ERROR)
    Jioqqo3" «Aulont

    printf("sendto failed with: Xd\n",
        WSAGetLastError());
    closesocket(sockM);
    closesocket(sock);
    WSACleanup();
    return -1;

> Sleep(500);
)

// Выход из группы путем закрытия sock. При использовании
// немаршрутизируемых плоскостей управления и данных WSAJoinLeaf TЭУ »(1
// возвращает тот же описатель сокета, который вы ей передали. t>Mlex>ld

closesocket(sock);

WSACleanup();
return -1;

```

## Пример многоадресной рассылки в АТМ-сети средствами Winsock 2

Листинг 11-3 содержит программу Mcastatm.c — простой пример многоадресной рассылки в сети АТМ, иллюстрирующий присоединение со стороны корня. Листинг не включает файл Support.c, который иницирует некоторые процедуры, используемые в примере, но не специфичные для многоадресной рассылки (например, *GetATMAddress*, которая просто возвращает АТМ-адрес локального интерфейса).

Вы увидите, что самая важная часть многоадресного корня — функция *Server*, которая использует цикл для вызова *WSAJoinLeaf* для каждого клиента, указанного в командной строке. Сервер хранит массив сокетов для каждого присоединяемого клиента, однако в вызовах *WSASend* используется главный сокет. Если бы протокол АТМ поддерживал эту возможность, сервер мог слушать на листовом соке (например, на соке, возвращенном *WSAJoinLeaf*), чтобы получать его «переписку». Другими словами, если клиент отправляет данные на связанном соке, сервер получает их на соответствующем описателе, возвращенном *WSAJoinLeaf*. Остальные клиенты не получают эти данные.

Посмотрите на функцию *Client*. Ее единственное требование к клиенту — создать привязку к локальному интерфейсу и ожидать приглашения от сервера в вызове *accept* или *WSAAccept*. После приема приглашения можно использовать новый сокет для получения данных от корня.



**Листинг 11 -3. Пример многоадресной рассылки в сети ATM (t) .S-r r**

```

// Модуль: Mcastatm.c                                     *tfnJhqa
                                                         'tee) U

<include "Support.h"

<include <stdio.h>
<include <stdlib.h>                                     зШ      tinea") tjn.hq

<define BUFSIZE                1024
<define MAX_ATM_LEAF          4

<define ATM_PORT_OFFSET        ((ATM_ADDR_SIZE * 2) - 2)
<define MAX_ATM_STR_LEN        (ATM_ADDR^SIZE * 2)

DWORD  dwAddrCount = 0,
        dwDataCount = 20;
BOOL    bServer = FALSE,
        bLocalAddress = FALSE;
char     szLeafAddresses[MAX_ATM_LEAF][MAX_ATH_STR_LEN + 1],
        szLocalAddress[MAX_ATM_STR_LEN + 1],
        szPort[3];
SOCKET sLeafSock[MAX_ATM_LEAF];

// Модуль: usage
//
// Описание:
//      Вывод информации об использовании
//
void usage(char *progrname)
{
    printf("usage: Xs [-s]\n", progrnae);
    printfC      -s      Act as root\n");
    printf("      -l:str      Leaf address to invite (38 chars)\n");
    printf("                  May be specified multiple times\n");
    printfC      -i:str      Local interface to bind to (38 chars)\n");
    printf("      -p:xx      Port number (2 hex chars)\n");
    printf("      -n:int      Number of packets to send\n");
    ExitProcessd);

// Модуль: ValidateArgs
//
// Описание:
//      Анализ параметров командной строки
//
void ValidateArgs(int argc, char **argv)
{
    i      int      i;

```

**Листинг 11-3.** *(продолжение)*

```

memset(szLeafAddresses, 0,                                \\
      MAX_ATM_LEAF * (HAX_ATM_STR_LEN + 1));
memset(szPort, 0, sizeof(szPort));                        us*qn TeyqNMqo* \\
                                                         "МЩВ ЙЮЮТЭП \\
for(i = 1; i < argc;                                       «jan яынонмдвоо \\
    if ((argv[i][0] == '-' || (argv[i][0] == '7'))
        switch (tolower(argv[i][1]))

    case 's':      // Сервер                                i SW      4UfLA8W
        bServer = TRUE;
        break;                                           isrto
    case "1":      // Адрес листа
        if (strlen(argv[i]) > 3)                          wb
            I *      strncpy(szLeafAddresses[dwAddrCount++],
                           &argv[i][3], MAX_ATM_STR_LEN - 2);
                                                         Jni
        break;
    case 'i':      // Локальный интерфейс wftRi. Mueqen    он «но3 \\
        if (strlen(argv[i]) > 3)
        {
            ^VO
            strncpy(szLocalAddress, &argv[i][3]/a         It
                  MAX_ATM_STR_LEN - 2);
            bLocalAddress = TRUE;                          - вя .
        }
    case 'p':      // Адрес порта                            {
        if (strlen(argv[i]) > 3)                            esie
            strncpy(szPort, &argv[i][3], 2);              JCTA
        break;                                              \4
    case 'n':      // Количество пакетов для отправки      щйонбгяЧ \\
        if (strlen(argv[i]) > 3)                            \\
            dwDataCount = atoi(&argv[i][3]);               • joinJe*)Но«
        break;                                              \\
    default:                                              лотЪ \\
        usage(argv[0]);                                     \x
        break;                                             ?<

}
return;

```

**Листинг 11-3. (продолжение)**

rtu(%KW\*MKWMM4K) ,6\*ft""

// Функция<sup>1</sup> Server

```

Ц Описание:
// Формирует привязку к локальному интерфейсу, а затем приглашает каждый;»?
// листовой адрес, указанный в командной строке. После установки
// соединения передает порцию данных.
II
void Server(SOCKET s, WSAPROTOCOL_INFO -lpSocketProtocol) {
    // Процедура сервера
    //
    SOCKADDR_ATM atmleaf, atmroot;
    WSABUF
    char sendbuf[BUFSIZE],
          szAddr[BUFSIZE];
    DWORD dwBytesSent,
           dwAddrLen = BUFSIZE,
           dwNumInterfaces,
           1;
    int ret;

    // Если не был указан определенный локальный интерфейс,
    // выбирает первый найденный
    //
    memset(&atmroot, 0, sizeof(SOCKADDR_ATM));
    if (!bLocalAddress)
    {
        dwNumInterfaces = GetNumATMInterfaces(s);
        GetATMAddress(s, 0, iatmroot.satm_number);
    }
    else
        AtoH(&atmroot.satm_number.Addr[0], szLocalAddress,
            ATM_ADDR_SIZE - 1);

    //
    // Установка номера порта в адресной структуре
    //
    AtoH(&atmroot.satm_number.Addr[ATM_ADDR_SIZE-1], szPort, 1);
    //
    // Заполнение оставшейся части структуры SOCKADDR_ATM
    //
    atmroot.satm_family = AF_ATM;
    atmroot.satm_number.AddressType = ATM_NSAP;
    atmroot.satm_number.NumofDigits = ATM_ADDR_SIZE;
    atmroot.satm_blli.Layer2Protocol = SAP_FIELD_ANY;
    atmroot.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
    atmroot.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
    //
    // Вывод на экран сведений о привязке и сама привязка

```

**Листинг 11-3.** (продолжение)

```

if (WSAAddressToString((LPSOCKADDR)&atmroot,
    sizeof(atmroot), IpSocketProtocol, szAddr,
    AdwAddrLen))

    printf("WSAAddressToString failed: Xd\n",
        WSAGetLastErrorO);

}

printf("Binding to: <Xs>\n", szAddr);

if (bind(s, (SOCKADDR *)&atmroot,
    sizeof(SOCKADDR_ATM)) == SOCKET_ERROR)
{
    printf("bind() failed: Xd\n", WSAGetLastErrorO);
    return;

>
// Приглашение каждого листа
//
for(i = 0; i < dwAddrCount; i++)
{

    // Заполнение структуры SOCKADDR_ATM для каждого листа
    //
    memset(&atmleaf, 0, sizeof(SOCKADDR_ATM));
    AtoH(&atmleaf.satm_number.Addr[0], szLeafAddresses[i]fb > t ;0 *
        ATM_ADDR_SIZE - 1);
    AtoH(&atmleaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
        szPort, 1);

    atmleaf.satm_family = AF_INET;
    atmleaf.satm_number.AddressType = ATM_INET;
    atmleaf.satm_number.NumofDigits = ATM_ADDR_SIZE;
    atmleaf.satm_blli.Layer2Protocol = SAP_FIELD_ANY;
    atmleaf.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
    atmleaf.satm.bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
}

// Вывод на экран клиентского адреса и его приглашение

if (WSAAddressToString((LPSOCKADDR)&atmleaf,
    sizeof(atmleaf), IpSocketProtocol, szAddr,
    &dwAddrLen))

    printf("WSAAddressToString failed: Xd\n",

        WSAGetLastErrorO);

printf("[X02d] Inviting' <Xs>\n", i, szAddr);

if ((sLeafSock[i] = WSAJoinLeaf(s,

```

Листинг 11-3. (продолжение)

```

(SOCKADDR -O&atmleaf, sizeof(SOCKADDR_ATH), NULL, addr4A3M)
NULL, NULL, NULL, J_SENDER_ONLY))
= INVALID_SOCKET)

printf("WSAJoinLeaf() failed: Xd\n",
      WSAGetLastError());
WSACleanup();
return;

>
// Заметьте: протокол ATM немного отличается от TCP.
// Когда завершается вызов WSAJoinLeaf,
// пользователь еще не обязательно принял соединение, поэтому немедленная
// отправка данных приведет к ошибке; так что ждем некоторое время, л»t

printf("Press a key to start sending.");
getchar();
printf("\n");

// Отправка порции данных групповому адресу, которая
// будет реплицирована всем клиентам

wsasend.buf = sendbuf;
wsasend.len = 128;
for(i = 0; i < dwOataCount; i++)
{
    memset(sendbuf, 'a' + (i X 26), 128);
    ret = WSASend(s, &wsasend, 1, &dwBytesSent, 0, NULL,
                  NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSASend() failed: Xd\n", WSAGetLastError());
        break;

        printf("[X02d] Wrote: Xd bytes\n", i, dwBytesSent);
        Sleep(500);

        for(i = 0; i < dwAddrCount; i++)
            closesocket(sLeafSock[i]);
        return;

>
//
// Функция: Client
//
// Описание:
// Сначала клиент привязывается к локальному интерфейсу
// (указанному в командной строке или к первому локальному ATM-адресу).

```

**Листинг 11-3.** {продолжение}

.\$-14

```
// Затем он ожидает приема приглашения, после чего ждет приема M
//
void Client(SOCKET s, WSAPROTOCOL_INFO lpSocketProtocol)

SOCKET si;
SOCKADDR_M atm_leaf, ,Xл4^4 us*. *. 1 ,i ti
atm_root; 03 <9-.
DWORD dwNumInterfaces, sJеOApW "л/бX ;бвШт (5i>fid")TrJnnq }
dwBytesRead,
dwAddrLen=BUFSIZE, п!лЭ1
dwFlags, {
WSA_BUF wsarecv;
char recvbuf[BUFSIZE], \ \
szAddr[BUFSIZE]; \ \
f ftX \ WSB w

int iLen = sizeof(SOCKADDR_ATM), « i a »
ret; •02 GUJAVMI = ^ .' (0

// Настройка локального интерфейса
II
memset(&atm_leaf, 0, sizeof(SOCKADDR_ATM));
if (IbLocalAddress) таавtowle3@fl")TjnJttq

dwNumInterfaces = GetNumATMInterfaces(s); вд «Htfqon ne«qft \ \
GetATMAddress(s, 0, &atm_leaf.satm_number); \ \
} 091 * tud vaeisew
else «to > 1 ;0 »
AtoH(&atm_leaf.satm_number.Addr[0], szLocalAddress, ;0 -
ATM_ADDR_SIZE-1); « i
AtoH(&atm_leaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
szPort, 1);

// Заполнение структуры SOCKADDR_ATM n} ti
// *
atm_leaf.satm_family = AF_ATM;
atm_leaf.satm_number.AddressType = ATM_NSAP;
atm_leaf.satm_number.NumofDigits = ATM_ADDR_SIZE;
atm_leaf.satm_bHli.Layer2Protocol = SAP_FIELD_ANY; :
atm_leaf.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
atm_leaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
// я
// Вывод на экран адреса, к которому мы привязываемся, и сама привязка
//
if (WSAAddressToString((LPSOCKADDR)&atm_leaf,
sizeof(atm_leaf), lpSocketProtocol, szAddr,
&dwAddrLen))
```

## Листинг 11-3. (продолжение)

```

    printf("CWSAAddressToString failed: %d\n",
           WSAGetLastError());

>
printf("Binding to %s\n", szAddr),

if (bind(s, (SOCKADDR *)&atm_leaf, sizeof(SOCKADDR_ATH))
    == SOCKET_ERROR)

    printf("bind() failed: %d\n", WSAGetLastError());
    return;

listen(s, 1);

// Ожидание приглашения

memset(&atm_root, 0, sizeof(SOCKADDR_ATM)),
if ((si = WSAAccept(s, (SOCKADDR *)&atm_root, &iilen, NULL*
0)) == INVALID_SOCKET)

    printf("WSAAccept() failed %d\n", WSAGetLastError());
    return;

printf("Received a connection\n");

// Прием порции данных

wsarecv.buf = recvbuf,
for(i = 0, l < dwDataCount;

    dwFlags = 0;
    wsarecv.len = BUFSIZE;
    ret = WSAREcv(sl, iwsarecv, 1, &dwBytesRead, &dwFlags,
        NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAREcv() failed: %d\n", WSAGetLastError());
        break,

        if (dwBytesRead == 0)
            break,
            recvbuf[dwBytesRead] = 0;
            printf("X02d] READ %d bytes: '%s'\n", i, dwBytesRead,
                recvbuf);

    }
    closesocket(sl),
    return;

```

Л-тHНrЪП,  
НО \*NёТbB \,

ТЗЗШ:  
JMOAXOO?

16

ш

W

rgibi

BT

f&gt;

?)g

4sa

V\

e&gt;raC W

\^

tee! mJ3

t&gt;eI\_e&gt;J&lt;&gt;

\\

ь т доa\*B \\

**Листинг 11-3.** (продолжение)

\м\ц\ \

// Функция: **mat**

тшт\*

//

) к

**Ц** Описание

```
// Эта функция загружает библиотеку Winsock, анализирует параметры
// командной строки, создает соответствующий сокет (с правильными
// корневыми или листовыми флагами) и запускает функции клиента или сервера
// в зависимости от заданных флагов.
```

```
int main(int argc, char **argv) {
```

```
    WSADATA          wsd;
    SOCKET            S;
    WSAPROTOCOL_INFO  lpSocketProtocol;
    DWORD             dwFlags;
```

```
    ValidateArgs(argc, argv);
```

```
    //
```

```
    // Загрузка библиотеки Winsock
```

```
    //
```

```
    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
```

```
    {
```

```
        printf("WSAStartup failed\n");
```

```
        return -1;
```

```
    }
```

```
    // Поиск ATM-совместимого протокола
```

```
    //
```

```
    if (FindProtocoK&lpSocketProtocol) == FALSE)
```

```
    {
```

```
        printf("Unable to find ATM protocol entryi\n");
```

```
        return -1;
```

```
    }
```

```
    // Создание сокета с соответствующими корневыми или листовыми флагами
```

```
    //
```

```
    if (bServer)
```

```
        dwFlags = WSA_FLAG_OVERLAPPED
```

```
            | WSA_FLAG_MULTIPOINT_C_ROOT
```

```
            | WSA_FLAG_MULTIPOINT_D_ROOT;
```

```
    else
```

```
        dwFlags = WSA_FLAG_OVERLAPPED
```

```
            | WSA_FLAG_MULTIPOINT_C_LEAF
```

```
            | WSA_FLAG_MULTIPOINT_D_LEAF;
```

```
    If ((s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
```

```
        FROM_PROTOCOL_INFO, &lpSocketProtocol, 0,
```

```
        dwFlags)) == INVALID_SOCKET)
```

```
        pnntf("socket failed with. Xd\n", WSAGetLastError0);
```

```
        WSACleanup0;
```

см. след. стр.



**Листинг 11 -3.**      {продолжение)

```

return - 1 ,

// Запуск соответствующего драйвера в зависимости от флагов,
// которые были заданы в командной строке

if (bServer)

    Server(s, SLP_SOCKET_PROTOCOL);

else

    Client(s, &IP_SOCKET_PROTOCOL);

closesocket(s)

WSACleanup();
return 0,
>

```

ЭМНЮШЬ  
5

АТАР  
П  
J03&W

## Общие параметры Winsock

Три параметра сокета применяются в обеих реализациях Winsock *IP\_MULTICAST\_TTL*, *IP\_MULTICAST\_IF* и *IP\_JMWULICAST\_LOOP*. Их обычно ассоциируют с параметрами Winsock 1 для присоединения и выхода из многоадресных групп, однако они могут в равной степени использоваться и в Winsock 2. Естественно, все три параметра относятся только к IP-рассылке.

### Параметр IP\_MULTICAST\_TTL

Задаёт значение TTL для данных рассылки. По умолчанию TTL равно 1, то есть данные отбрасываются первым же маршрутизатором и доставляются лишь по своей сети. Если вы увеличите TTL, данные смогут пересечь столько маршрутизаторов, сколько указано в TTL. Маршрутизатор не пересылает дейтаграммы с адресами назначения в диапазоне 224.0.0.0 — 224.0.0.255, независимо от значения TTL. Эта область адресов зарезервирована для протоколов маршрутизации и других низкоуровневых протоколов изучения топологии, а также служебных протоколов — типа поиска шлюзов и сообщения о членстве в группе.

Когда вы вызываете *setsockopt*, параметр *level* должен быть равен *IPPROTO\_IP*, *optname* — *IP\_MULTICAST\_TTL*, а *optval* — содержать целое число, задающее значение TTL. Следующий фрагмент иллюстрирует настройку TTL.

```

int    optval,
optval = 8,
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&optval,
    sizeof(mt)) == SOCKET_ERROR)
<
// Ошибка

```

5 ^  
>

Помимо этого параметра можно использовать *SIO\_MULTICAST\_SCOPE* с функцией *WSAIoctl* или *ioctlsocket*

Параметр TTL не обязателен для многоадресной рассылки в ATM, так как отправка данных через ATM производится в одном направлении и все получатели известны. Поскольку плоскость управления — маршрутизируемая, узел *s\_root* должен явно приглашать каждый лист. Так что вам не нужно ограничивать область передачи данных — данные не будут дублироваться в сетях, где может не оказаться участников многоадресной рассылки.

### Параметр *IP\_MULTICAST\_IF*

Этот параметр задает IP-интерфейс, с которого рассылаются данные. Обычно маршрут (не в случае многоадресной рассылки) для интерфейса, с которого отправляется дейтаграмма, задает таблица маршрутов. Система сама определяет наилучший интерфейс для конкретной дейтаграммы и ее направления. Впрочем, поскольку групповые адреса могут быть использованы кем угодно, таблицы маршрутов не достаточно. Программист должен знать, куда направляются данные. Конечно, это необходимо, только если компьютер, на который отправляются данные рассылки, подключен к нескольким сетям через несколько сетевых адаптеров. В этом случае параметр *optval* содержит адрес локального интерфейса, на который надо отправлять данные рассылки.

```
DWORD    dwInterface,

dwInterface = inet_addr( "129.121.32.19"),
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)dwInterface,
    sizeof(DWORD)) == SOCKET_ERROR)
{
    // Ошибка
}
```

В этом примере мы задали для локального интерфейса адрес 129.121.32.19. Любые данные, отправленные на сокет 5, передаются с сетевого интерфейса, которому назначен этот IP-адрес.

ATM не требует отдельного параметра сокета для задания интерфейса. Можно явно привязать *s\_root* к конкретному интерфейсу перед вызовом *WSAJonLeaf*. Аналогично, клиент должен быть связан с конкретным ATM-интерфейсом, чтобы ожидать приглашения, вызывая *accept* или *WSAAccept*.

### Параметр *IP\_MULTICAST\_LOOP*

Последний параметр определяет, будет ли приложение получать данные собственной рассылки. Если приложение присоединяется к многоадресной группе и отправляет данные в группу, оно тоже получит эти данные. Если в ходе отправки данных есть ожидающий выполнения вызов *recvfrom*, он вернет копию данных. Заметьте, для отправки данных в многоадресную группу не нужно присоединять к ней приложение. Это требуется, только если вы хотите получать данные, адресованные группе. Параметр сокета *IP\_MULTICAST\_LOOP* предназначен для отключения эха данных на локальный интерфейс. В качестве пара-

метра *optval* передайте целое число, которое является логическим значением и определяет, включить или отключить петлю ф

```
int    optval,
го
optval = 0,      // Отключение петли я*
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&optval, "v
    sizeof(int)) == SOCKET.ERROR) m,
{ i
    // Ошибка
}
```

Те же функции выполняет *ioctl*-команда *SIO\_MULTIPOINT\_LOOPBACK* вместе с *WSAIoctl* или *ioctlsocket*. К сожалению, этот параметр сокета не реализован в Windows 95, Windows 98 и Windows NT 4 и по умолчанию петля включена. Если вы попытаетесь отключить ее таким способом, вернется ошибка *WSAENOPROTOOPT*.

По определению корневой узел ATM — единственный, которому разрешено отправлять данные в многоадресную группу, — не будет получать свои собственные данные, так как корневой сокет не является листовым, а получать данные многоадресной рассылки в сети ATM могут только листовые сокеты. Тот же процесс, который создает *c\_root*, может создать и отдельный узел *c\_leaf*, который *c\_root* затем пригласит. Впрочем, это будет «искусственная» петля.

## Ограничение многоадресной рассылки \_ при удаленном доступе

При попытке отправить или получить данные рассылки через *интерфейс службы удаленного доступа* (Remote Access Service, RAS) вы можете столкнуться с одним ограничением. Фактически оно связано с сервером, к которому вы подключаетесь. Большинство Windows-серверов удаленного доступа работают под управлением Windows NT 4, в которой нет IGMP-прокси. Это означает, что любые запросы на присоединение к группе не вернутся от сервера Windows NT 4. В результате приложение не сможет подключиться к группе, а значит — отправлять или получать данные рассылки. В RAS-сервере для Windows 2000 есть IGMP-прокси, хотя по умолчанию он выключен. Но после его включения удаленные клиенты будут подключаться к группам, а также отправлять и получать многоадресные данные.

## Резюме

Многоадресная рассылка оптимизирует работу приложений, которым необходимо поддерживать связь с множеством конечных точек без издержек, характерных для широковещания. В этой главе мы дали определение многоадресной рассылке и рассмотрели ее модели. Затем обсудили, как многоадресная рассылка в сетях IP и ATM применяется к этим моделям. И, наконец, остановились на том, как многоадресная рассылка реализуется в Winsock 1 через параметры сокетов и в Winsock 2 — с помощью функции *WSAJoinLeaf*.

## Качество обслуживания

С появлением разнообразных мультимедийных приложений и растущей популярностью Интернета пропускная способность многих сетей снижается до критических отметок. Эта проблема особо актуальна в сетях с общей средой передачи, таких как Ethernet, поскольку в них весь трафик обрабатывается одинаково, и даже одно приложение может переполнить сеть. *Качество обслуживания* (Quality of Service, QoS) — это набор компонентов, допускающий дифференциацию и предпочтительную обработку данных в сети QoS-совместимая сеть.

*III* предотвращает злоупотребления сетевыми ресурсами со стороны неадаптируемых протоколов (таких, как UDP),

*III* оптимально распределяет ресурсы между *негарантированным* (best-effort) трафиком, а также трафиком с высоким или низким приоритетом,

- резервирует ресурсы для правомочных пользователей,

*И* определяет очередность доступа к ресурсам для разных пользователей

Generic Quality of Service (GQoS) — это реализация QoS фирмой Microsoft. В настоящее время Microsoft поставляет QoS-совместимый поставщик TCP/IP и UDP/IP в составе Windows 98 и Windows 2000. Протокол ATM также допускает использование службы QoS, поскольку она встроена в него изначально.

В этой главе рассказывается о службе QoS и ее реализации на платформах Win32. Будут рассмотрены компоненты, необходимые для предпочтительной обработки сетевого трафика. Мы также покажем, как средствами Winsock написать сетевое приложение, использующее преимущества этих компонентов для повышения скорости и резервирования пропускной способности. Большая часть главы посвящена службе QoS в IP-сетях, а последние разделы — немного отличающейся от нее QoS в ATM-сетях.

**ПРИМЕЧАНИЕ** На протяжении всей главы служба Quality of Service будет называться QoS. Предполагается, что обсуждается реализация QoS от Microsoft.

## Введение

Для работы QoS требуются

- **устройства в сети**, такие, как маршрутизаторы и коммутаторы, знакомые с этой дифференциацией **служб**;

- **локальные рабочие станции**, способные определять приоритетность трафика, который они помещают в сеть;
- **компоненты политики безопасности**: кому и в каком размере разрешено использовать имеющуюся пропускную способность.

Но прежде чем приступить к изучению этих компонентов, рассмотрим *протокол резервирования ресурсов* (Resource Reservation Protocol, RSVP) — сигнальный протокол, используемый для связи между QoS-отправителями и QoS-приемниками.

## Протокол RSVP

Протокол RSVP связывает воедино компоненты сети, приложения и параметры политики безопасности. Он передает запросы резервирования ресурсов по сети, которая может состоять из различных сред передачи. RSVP передает QoS-запросы пользователей всем сетевым устройствам, позволяя им резервировать ресурсы. В результате узлы сети указывают, отвечает ли сеть нужному уровню обслуживания.

RSVP резервирует сетевые ресурсы, задавая сквозные *потоки* (flows) в сети. Поток — это сетевой путь, связанный с одним или несколькими отправителями, одним или несколькими приемниками и определенным уровнем QoS. Узел, отправляющий данные, которые нуждаются в определенном уровне обслуживания, отправляет предполагаемому приемнику или приемникам сообщение PATH с требованиями к пропускной способности. Соответствующие параметры передаются по пути предполагаемым приемникам.

Принимающий узел, заинтересованный в этих данных, резервирует ресурсы для потока (и весь путь от отправителя), посылая отправителю сообщение RESV (резервировать). После этого промежуточные RSVP-устройства определяют, могут ли они выполнить требования к пропускной способности и имеет ли пользователь право запрашивать эти ресурсы. Если ответ в обоих случаях положительный, каждое из устройств резервирует ресурсы и передает отправителю сообщение RESV.

Когда отправитель получает сообщение RESV, начинается передача данных QoS. Периодически каждая конечная точка внутри потока посылает сообщения PATH и RESV, чтобы подтвердить резервирование и получить сетевую информацию об изменении пропускной способности. Периодическое обновление сообщений PATH и RESV также позволяет протоколу RSVP оставаться динамичным и «на лету» выявлять лучшие (например, более быстрые) маршруты. При обсуждении Winsock далее в этой главе мы вернемся к RSVP и рассмотрим, как вызовы API-функций Winsock его иницииируют.

Заметьте: это одностороннее резервирование, даже если приложение запрашивает пропускную способность и для передачи, и для приема. Один сеанс иницииируется для требований передачи, другой — для требований приема.

## Сетевые компоненты

Для работы QoS необходимо, чтобы сетевые устройства между двумя конечными точками различали приоритеты трафика. Тогда они смогут направлять

трафик так, чтобы выполнялась гарантия QoS, полученная приложением. Кроме того, эти сетевые устройства должны уметь определять, достаточно ли пропускная способность сети, когда приложение ее запрашивает. Это обеспечивают:

**III 802.1p** — стандарт, назначающий приоритет пакетов в подсети, путем задания трех битов внутри MAC-заголовка пакетов;

**III IP-приоритет** — метод задания очередности IP-пакетов;

**III** оповещение на канальном уровне — механизм привязки RSVP-объектов к QoS-компонентам ГВС на втором уровне сетевой модели OSI;

- **Subnet Bandwidth Manager (SBM)** — диспетчер, управляющий пропускной способностью в сети с общей средой передачи;

**К Resource Reservation Protocol (RSVP)** — протокол, передающий QoS-запросы и информацию QoS-совместимым сетевым устройствам по пути между отправителем и одним или несколькими приемниками.

### Стандарт 802.1p

Соблюдение гарантий QoS и неравноправная обработка пакетов возлагается в основном на концентраторы и коммутаторы сети. Они находятся на канальном уровне модели OSI и располагают информацией о полях только внутри заголовка *управления доступом к среде* (media access control, MAC) в начале каждого пакета.

802.1p — это стандарт, задающий очередность обработки пакетов путем присвоения трехбитного значения приоритета в заголовке MAC. Когда подсеть сети, отличной от 802.1p, становится перегруженной, коммутаторы и маршрутизаторы не справляются с трафиком, и образуется задержка. В 802.1p-сетях коммутаторы и маршрутизаторы задают очередность входящего трафика, основываясь на битах приоритета, и в первую очередь обрабатывают пакеты с более высоким приоритетом.

Внедрение стандарта 802.1p для QoS требует специального оборудования, способного распознавать это трехбитное поле: *платы сетевого интерфейса* (network interface card, NIC), сетевых драйверов и коммутаторов.

### IP-приоритет

IP-приоритет — это метод определения значений приоритета на более высоком уровне, чем 802.1p. Он позволяет дифференцировать относительные приоритеты пакетов, проходящих через устройства сетевого уровня OSI (например, маршрутизаторы). IP-приоритет реализует поле *типа службы* (type of service, TOS) внутри IP-заголовка, которое задает разные уровни приоритета. На основании этих битов маршрутизаторы назначают очереди приоритетов, в результате чего трафик более высокого приоритета обслуживается в первую очередь.

Как и в случае 802.1p, для работы IP-приоритета все устройства сетевого уровня в сети должны распознавать значения битов IP-приоритета и соответствующим образом управлять трафиком.

## Оповещение на канальном уровне

Оповещение на канальном уровне необходимо, когда трафик перемещается по ГВС. Обычно ГВС связывает несколько сетей и таким образом по мере передачи данных управляет информацией на физическом, канальном и сетевом уровнях. Для обеспечения сквозного QoS-соединения ГВС должна распознавать очередность трафика QoS. Для этого QoS обеспечивает привязку параметров RSVP и других параметров QoS к естественному для ГВС базовому способу оповещения на канальном уровне, с помощью которого ГВС-технологии реализуют свою собственную службу QoS.

## Диспетчер SBM

Этот диспетчер управляет ресурсами в сети с общей средой передачи, например, в Ethernet, а также качеством обслуживания QoS-приложений. SBM необходим в таких сетях, поскольку когда конечная точка запрашивает QoS для приложения, каждое сетевое устройство принимает или отвергает запрос в зависимости от выделенных этому устройству частных ресурсов. Сетевые устройства не знают о доступности ресурсов в общей среде передачи. SBM решает эту проблему, становясь брокером для таких устройств. SBM также тесно связан со *службой управления допуском* (Admission Control Service, ACS), являющейся частью политики безопасности. SBM проверяет, имеет ли приложение (или пользователь) право запрашивать пропускную способность. Заметьте: SBM для сети может быть узлом под управлением Windows 2000 Server.

## Компоненты приложения

Рассмотрим, как локальная система задает очередность данных на основе уровней QoS, которые запросило приложение. Чтобы локальная система поддерживала QoS, необходимы:

- **поставщик службы GQoS** — обращается к другим QoS-компонентам;
- Я модуль управления трафиком (Traffic Control module, TC)** — управляет исходящим трафиком; включает *классификатор пакетов* (Generic Packet Classifier), *планировщик пакетов* (Packet Scheduler) и *формировщик трафика* (Packet Shaper);
- III протокол резервирования ресурсов (Resource Reservation Protocol, RSVP)** — протокол, вызываемый компонентом службы GQoS и передающий по сети запрос на резервирование;
- **GQoS API** — программный интерфейс для GQoS, такой как Winsock;
- **Traffic Control (TC) API** — программный интерфейс для компонентов управления трафиком, регулирующий трафик на локальном узле.

## Поставщик службы GQoS

Этот компонент влияет почти на все функции QoS: иницирует модуль TC и реализует, поддерживает и управляет оповещением RSVP для работы GQoS.

Для поиска на узле QoS-совместимых поставщиков можно запросить каталог поставщика с помощью функции *WSAEnumProtocols*. Флаг совместимости с QoS находится в структуре *WSAPROTOCOLINFO*, возвращаемой *WSAEnumProtocols*, — *dwServiceFlags1* (флаг для проверки *XPI\_QOS\_SUPPORTED*). Подробнее о *WSAEnumProtocols* — в главе 5.

## Модуль TC

Модуль TC играет главную роль в QoS, поскольку задает приоритет пакетов как внутри, так и снаружи узла сети, на котором активизирован. Результаты этой предпочтительной обработки пакетов, проходящих через систему и сеть, ощущаются по всей сети и напрямую влияют на характеристики QoS. Модуль TC реализуют Generic Packet Classifier, Packet Scheduler и Packet Shaper.

**Классификатор пакетов Generic Packet Classifier (GPC).** В его обязанности входит классификация и задание очередности пакетов внутри сетевых компонентов. GPC также задает приоритеты для таких задач, как время использования центрального процессора или передача в сети путем создания таблиц поиска и служб классификации в сетевом стеке. Это первый этап процесса задания очередности для сетевого трафика.

**Планировщик пакетов Packet Scheduler** определяет способ передачи данных, выполняя одну из ключевых функций QoS. Это модуль управления трафиком, регулирующий, сколько данных может получить приложение или поток. Планировщик использует схему задания очередности, предлагаемую GPC, и обеспечивает разные уровни обслуживания для трафика разного приоритета. Данные, которым GPC присвоил более высокий приоритет, обрабатываются в первую очередь.

**Формировщик трафика Packet Shaper** регулирует передачу данных от потоков в сеть. Большинство приложений считывает и записывает данные в виде пакетов. Между тем, многим QoS-приложениям необходима определенная скорость передачи данных. Формировщик планирует передачу данных в единицу времени, сглаживая нагрузку на сеть.

## Интерфейс Traffic Control API

Этот интерфейс регулирует сетевой трафик на локальном узле и включает методы управления QoS-компонентами — GPC, планировщиком и формировщиком. Некоторые функции Traffic Control выполняются неявно через вызовы к GQoS-совместимым функциям Winsock, которые обрабатывает поставщик службы GQoS. Впрочем, приложения, напрямую управляющие компонентами TC, могут сделать это с помощью API-функций TC, которые в этой книге не рассматриваются (подробности см. в Platform SDK). API-функции Winsock GQoS мы рассмотрим далее в этой главе.

## Компоненты политики безопасности

Этот третий и последний компонент GQoS управляет выделением ресурсов QoS-совместимым приложениям. Компоненты политики безопасности наиболее интересны системным администраторам, которые полномочны выделять



ресурсы, в том числе пропускную способность, конкретным пользователям или определенным приложениям Среди этих компонентов

**III служба управления допуском (Admission Control Service, ACS)** — служба Windows 2000 Server, перехватывающая RSVP-сообщения PATH и RESV для управления доступом QoS-совместимых клиентов к разным уровням гарантий, предоставляемых средствами QoS,

**К модуль локальной политики (Local Policy Module, LPM)** — управляет доступом к ресурсам на основе политик, заданных в ACS для SBM,

**III элемент политики (Policy Element, PE)** — располагается на клиенте и выдает аутентификационную информацию для запросов на резервирование

## Служба ACS

Эта служба регулирует использование сети QoS-совместимыми приложениями посредством RSVP-протокола ACS перехватывает сообщения PATH и RESV, чтобы проверить, имеет ли запрашивающее приложение достаточные привилегии После перехвата RSVP-сообщение передается модулю LPM, выполняющему аутентификацию

ACS располагается на компьютере с Windows 2000 и настраивается системным администратором, который задает ограничения ресурсов для пользователей, приложений или групп

## Модуль локальной политики безопасности

После перехвата RSVP-сообщения и добавления информации о пользователе ACS передает их модулю LPM LPM ищет запись о пользователе в Active Directory, проверяя сведения Если сетевые ресурсы доступны (как определил SBM) и аутентификация пройдена успешно, сообщение RSVP, перехваченное ACS, пересылается далее Если пользователь не имеет прав на запрос определенного уровня QoS, генерируется указывающая на это ошибка, которая возвращается внутри RSVP-сообщения

**ПРИМЕЧАНИЕ** Учетные записи пользователей должны быть частью домена Windows 2000

## Элемент политики

PE фактически содержит сведения о политике безопасности, которые запрашивает модуль LPM Эти структуры данных не рассматриваются в нашей книге, поскольку в основном имеют отношение к администрированию сетевых ресурсов

## QoS и Winsock

Для программного доступа к QoS из приложений применяется Winsock 2 Прежде всего мы рассмотрим структуры QoS верхнего уровня, необходимые большинству вызовов Winsock, затем — функции Winsock, вызывающие QoS

на сокете, и завершение QoS, активизированного на сокете. После этого обсудим объекты, которые можно использовать для управления работой поставщика службы QoS или возвращения информации от него.

Переход от обсуждения основных структур QoS к функциям QoS и обратно к структурам, относящимся к поставщику, может показаться нелогичным. И все же перед тем как рассмотреть работу поставщика, мы хотим подробно объяснить, как основные структуры взаимодействуют с API-вызовами Winsock.

## Структуры QoS

Структура *QoS* — основа программирования QoS. Она состоит из

**III структуры *FLOWSPEC***, описывающей уровни QoS, используемые приложением отдельно для передачи и для приема данных,

**Ж буфера поставщика службы**, содержащего характеристики QoS для данного поставщика.

### Структура QOS

Структура *QOS* задает параметры **QoS** для передачи и приема трафика.

```
typedef struct _QualityOfService
{
    FLOWSPEC    SendingFlowspec,
    FLOWSPEC    ReceivingFlowspec,
    WSABUF      ProviderSpecific,
} QOS, FAR * LPQOS;
```

Структуры *FLOWSPEC* определяют характеристики и требования для каждого направления трафика. Поле *ProviderSpecific* используется для возвращения информации и для изменения рабочих параметров QoS.

### Структура FLOWSPEC

*FLOWSPEC* — основная структура, описывающая отдельный поток (напомним, поток описывает данные, перемещаемые в одном направлении).

```
typedef struct _flowspec
{
    ULONG      TokenRate,
    ULONG      TokenBucketSize;
    ULONG      PeakBandwidth,
    ULONG      Latency,
    ULONG      DelayVariation,
    SERVICE_TYPE ServiceType,
    ULONG      MaxSduSize,
    ULONG      MinimumPolicedSize,
} FLOWSPEC, *PFLOWSPEC, FAR *LPFLOWSPEC;
```

Рассмотрим каждое из полей структуры *FLOWSPEC*

**Поле *TokenRate*** задает скорость передачи данных в байтах в секунду. Если по какой-то причине скорость передачи снижается, приложение может накопить дополнительные блоки данных (*token*), чтобы передать их позже. Количество таких блоков ограничивается значением *PeakBandwidth*, а накопление самих блоков (так называемая корзина блоков) — значением поля *TokenBucketSize*. Это суммарное ограничение позволяет избежать появления неактивных потоков, которые накопили слишком много данных, если бы они вдруг выдали в сеть собранные данные, емкость канала связи была бы исчерпана. Для управления трафиком и соблюдения целостности ресурсов сетевых устройств потокам разрешено накапливать данные (на скорости *TokenRate*) только до значения своего поля *TokenBucketSize*. Кроме того, < пакетные > передачи разрешены только в объемах, ограниченных значением *PeakBandwidth*.

Управление трафиком поддерживается за счет того, что потоки не могут отправлять слишком много данных за один раз, а целостность ресурсов сетевых устройств — за счет того, что такие устройства запасают пакеты с высоким приоритетом.

Из-за этих ограничений приложение может начать передачу, только если накоплено достаточно данных. Иначе приложение либо ожидает поступления дополнительной информации, либо полностью отбрасывает данные. Модуль ТС определяет, что происходит с данными, слишком долго ожидающими передачи. Поэтому приложение должно следить, чтобы значение поля *TokenRate* было разумным. Если приложение не требует планирования скорости передачи, этому полю можно присвоить значение *QOSJ4OT SPECIFIED (-1)*.

**Поле *TokenBucketSize***. Значение этого поля ограничивает количество данных, которые могут накапливаться для определенного потока. Например, видеоприложения обычно задают этому полю размер передаваемого кадра, поскольку, как правило, им требуется передавать только целые видеокдры. Приложения, требующие постоянной скорости передачи данных, присвоят этому полю значение, допускающее некоторые вариации. Значение поля *TokenBucketSize*, как и *TokenRate* выражается в байтах в секунду.

**Поле *PeakBandwidth***. Определяет максимальное количество данных, передаваемое за указанный период времени. Фактически это значение задает максимальный объем «пакетной» передачи — это очень важно, поскольку не позволяет приложениям, накопившим много данных, переполнить ими сеть. *PeakBandwidth* выражается в байтах в секунду.

**Поле *Latency*** задает максимально допустимую задержку между передачей бита и его приемом предполагаемым адресатом. Интерпретация этого значения зависит от уровня обслуживания, запрошенного в поле *ServiceType*. Задержка выражается в микросекундах.

**Поле *DelayVariation*** определяет разницу между минимальной и максимальной задержкой доставки пакета. Обычно приложение использует это значение для определения размера буфера приема данных и поддержки исходных условий передачи. *DelayVariation* выражается в микросекундах.

Поле *ServiceType* определяет уровень обслуживания, необходимый потоку данных. Могут быть заданы следующие типы обслуживания:

**III *SERVICETYPENOTRAFFIC*** — указывает, что в этом направлении данные не передаются.

**III *SERMCETYPEBESTEFFORT*** — указывает, что параметры, заданные в структуре *FLOWSPEC*, рекомендательные, и что система попытается поддержать этот уровень обслуживания. Доставка пакета не гарантируется.

**III *SERVICETYPECONTROLLEDLOAD*** — указывает, что параметры передачи должны вплотную приближаться к обеспечиваемым при негарантированном обслуживании в не загруженной трафиком сети. Это предполагает два условия. Во-первых, потери пакетов будут примерно соответствовать обычному уровню ошибок среды передачи, во-вторых, задержка передачи не намного превысит минимальную.

- ***SERVICETYPEGUARANTEED*** — гарантирует в течение всего соединения передачу данных со скоростью, заданной в поле *TokenRate*. Если фактическая скорость передачи данных превысит значение *TokenRate*, данные МОГУТ быть задержаны или отброшены (в зависимости от настройки ТС). Если же значение *TokenRate* не превышено, соблюдение параметров задержки (*Latency*) также гарантировано.

Помимо этих четырех типов обслуживания есть несколько других флагов, которые могут вернуть приложению полезную информацию. Над этими информационными флагами и любым действительным флагом *ServiceType* можно выполнить логическую операцию ИЛИ. Вот эти флаги, называемые флагами модификатора типа обслуживания:

- ***SERVICETYPENETWORKUNAVAILABLE*** — несоблюдение условий обслуживания в направлении передачи или приема.

**III *SERMCETYPEGENERALINFORMATION*** — для потока поддерживаются все типы обслуживания.

**K *SERVICETYPENOCHANGE*** — запрашиваемый уровень обслуживания QoS не изменился. Этот флаг может быть возвращен из вызова *Winsock*, или приложение указывает его при повторном согласовании параметров QoS, сообщая, что в данном направлении уровни QoS не изменились.

« ***SERMCEIMMEDIATE TRAFFIC CONTROL*** — приложение использует этот флаг, чтобы немедленно вызвать ТС вместо негарантированной передачи до приема сообщения *RESV*.

**Я *SERVICENOTRAFFICCONTROL*** — над этим и другими флагами *ServiceType* можно выполнить логическую операцию ИЛИ, чтобы полностью отключить управление трафиком.

- ***SERMCENOQOSSIGNALING*** — этот флаг можно использовать вместе с предыдущим, чтобы предотвратить передачу любых оповещающих *RSVP*-сообщений. Будет вызван локальный компонент управления трафиком, но *RSVP*-сообщения *PATH* не отправятся. Этот флаг может также использоваться совместно с принимающей структурой *FLOWSPEC* для подавле-

ния автоматической генерации сообщения RESV Приложение получает уведомление, что сообщение PATH поступило, после чего изменяет QoS, вызывая *WSAioctl (SIO\_SET\_QOS)*, чтобы сбросить этот флаг и тем самым выдать сообщение RESV

**Поле *MaxSduSize*** задает максимальный размер пакета данных, передаваемых в определенном потоке Выражается в байтах

**Поле *MinimumPolicedSize*** задает минимальный размер пакета данных, передаваемых в определенном потоке Выражается в байтах

## Функции, вызывающие QoS

Предположим, приложение запрашивает определенную пропускную способность сети Этот процесс инициируют четыре функции После начала сеанса RSVP приложение регистрируется для получения событий *FD\_QOS* Информация о состоянии QoS и коды ошибок передаются ему, как события *FD\_QOS* Приложение может зарегистрироваться для приема этих событий обычным способом включить флаг *FD\_QOS* в поле события функции *WSAAsyncSelect* или *WSAEventSelect*

Уведомление *FD\_QOS* особенно существенно, если соединение устанавливается с помощью структур *FLOWSPEC* со значениями по умолчанию (*QOS\_NOT\_SPECIFIED*) После того как приложение сделало запрос QoS, базовый поставщик периодически обновляет структуру *FLOWSPEC* для указания текущего состояния сети и уведомляет приложение, выдавая событие *FD\_QOS* Располагая этой информацией, приложение может запросить или изменить уровни QoS, чтобы отобразить имеющуюся пропускную способность Помните обновленная информация указывает только на локально доступную пропускную способность и не обязательно сообщает о сквозной пропускной способности

После создания потока сетевая пропускная способность может меняться, или отдельный участник потока может изменить запрошенный уровень обслуживания Повторное согласование выделенных ресурсов вызывает генерацию события *FD\_QOS*, которое сообщает об этих изменениях приложению На этом этапе приложение должно вызвать *SIO\_GET\_QOS*, чтобы получить новые уровни ресурсов Оповещения о событиях QoS и обмен информацией о состоянии мы обсудим далее, в разделе, посвященном программированию QoS

## Функция *WSAConnect*

Клиент использует функцию *WSAConnect*, чтобы инициировать одноадресное QoS-соединение с сервером Запрашиваемые значения QoS передаются как параметры *ipSQOS* В настоящее время групповое QoS не поддерживается и не реализовано, *ipGQOS* должно передаваться нулевое значение

```
int WSAConnect (
    SOCKET s,
    const struct sockaddr FAR <name,
    int namelen,
```

```

LPWSABUF lpCallerData,
LPWSABUF lpCalleeData,
LPQOS lpSQOS,
LPQOS lpGQOS

```

Вызов *WSAConnect* может использоваться совместно с требующими и не требующими соединения сокетами. В первом случае эта функция устанавливает соединение и генерирует соответствующие сообщения PATH и (или) RESV.

При использовании не требующих соединения сокетов необходимо связать адрес конечной точки с сокетом, чтобы поставщик службы знал, куда отправлять сообщения PATH и RESV. Важно учесть, что только данные, отправленные на адрес приемника, будут обработаны системой согласно уровням QoS, заданным для этого сокета. Другими словами, если *WSAConnect* используется для сопоставления конечной точки не требующему соединения сокету, данные будут передаваться только между этими двумя конечными точками в течение времени жизни этого сокета. Если необходимо передать данные с гарантиями QoS нескольким конечным точкам, используйте *WSAIoctl* или *SIO\_SET\_QOS* для указания всех нужных конечных точек.

### Функция *WSAAccept*

Функция *WSAAccept* принимает соединение клиента, которое может поддерживать QoS.

```

SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR «addr,
    LPINT  addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD  dwCallbackData

```

Если вам требуется условная функция, ее прототип должен выглядеть так:

```

int CALLBACK ConditionalFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR *g,
    DWORD dwCallbackData

```

Недостаток в том, что поставщик не гарантирует возвращение действительных значений QoS, которые клиент запрашивает как параметр *ipSQoS*. Так что для активизации QoS на клиентском сокете должна быть вызвана функция *WSAIoctl* с параметром *SIO\_SET\_QOS* до или после вызова функции *WSAAccept*. Если QoS включено на прослушивающем сокет, эти значения будут скопированы на клиентский сокет по умолчанию.

На самом деле, условная функция бесполезна. При использовании протокола TCP нельзя отклонить клиентское соединение само по себе, потому что к моменту вызова условной функции оно уже было установлено на уровне TCP. Кроме того, поставщик не передаст действительные параметры QoS условной функции, даже если сообщение PATH уже поступило. В силу этих причин мы не рекомендуем применять условную функцию *WSAAccept*.

**ПРИМЕЧАНИЕ** При использовании *WSAAccept* в Windows 98 необходимо учесть следующее. Если вы используете условную функцию с *WSAAccept* и значение параметра *IpSQOS* не пустое, включите QoS (используя *SIO\_SET\_QOS*). Иначе *WSAAccept* не завершится успешно.

## Функция *WSAJoinLeaf*

Эта функция используется для многоточечных соединений (подробнее о многоадресном вещании — в главе 11):

```
SOCKET WSAJoinLeaf(
    SOCKET s,
    const struct sockaddr FAR <name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    DWORD dwFlags
);
```

Чтобы приложение могло присоединиться к сеансу групповой рассылки, оно должно создать сокет с соответствующими флагами (*WSA\_FLAG\_MULTICAST\_ROOT*, *WSA\_FLAG\_MULTICAST\_C\_LEAF*, *WSA\_FLAG\_MULTICAST\_D\_ROOT* и *WSA\_FLAG\_MULTICAST\_D\_LEAF*). Когда приложение устанавливает многоточечные соединения, оно задает параметры QoS в параметре *IpSQOS*.

При использовании *WSAJoinLeaf* операция присоединения к многоадресной группе IP отделена от установки сеанса QoS RSVP. Вероятно, подключение к многоадресной группе будет успешным. Функция возвращается, не завершив запрос на резервирование. Позднее вы получите сообщение *FD\_QOS* об успехе или неудаче выделения запрошенных ресурсов.

Помните о времени жизни (TTL), указанном для многоадресных данных. Если вы планируете задать TTL с параметрами *SIO\_MULTICAST\_SCOPE* или *IP\_MULTICAST\_TTL*, это следует сделать до вызова *WSAJoinLeaf* или *ioctl*-команды *SIO>ET\_QOS* для включения QoS на сокете. Если область действия определена после включения QoS, указанное TTL не вступит в силу до повторного согласования параметров QoS посредством *SIO\_SET\_QOS*. Значение TTL также будет передаваться запросом RSVP.

Важно настроить TTL до включения QoS на сокете, поскольку TTL многоадресной рассылки, заданное на сокете, также влияет на время жизни сообщений RSVP, от чего напрямую зависит количество сетей, которым будет передан запрос на резервирование ресурсов. Например, чтобы задать несколь-

ко конечных точек в многоадресной группе IP, охватывающей три сети, присвойте TTL значение 3, чтобы генерируемый сетевой трафик не передавался «лишним» сетям. Если TTL не настроить до вызова *WSAJomLeaf*, сообщения RSVP будут отправляться со стандартным временем жизни (63), из-за чего узел попытается зарезервировать ресурсы в чрезмерном количестве сетей.

### Функция *WSAIocctl*

Функция *WSAIocctl* с *ioctl*-параметром *SIO\_SET\_QOS* используется либо для первого запроса QoS на сокете, либо для повторного согласования условий QoS после их начального запроса. При ее использовании в случае неудачи запроса QoS возвращается более подробная информация об ошибке в сведениях от поставщика. (Подробнее о функции *WSAIocctl* и ее вызове, а также о параметрах *SIO\_SET\_QOS* и *SIO\_GET^QOS* - в главе 9.)

С помощью параметра *SIO\_SET\_QOS* задают или изменяют параметры QoS на сокете. Используя *WSAIocctl* с *SIOSET\_QOS*, можно определять объекты, относящиеся к поставщику, для тонкой настройки работы QoS. В частности, приложение, где используются не требующие соединения сокеты и не применяется *WSAConnect*, может вызвать *WSAIocctl* с *SIO\_SET\_QOS* и указать объект адреса назначения в буфере поставщика, чтобы определить конечную точку, пригодную для установления сеанса RSVP. При задании параметров QoS передайте структуру *QOS* как *ipvInBuffer* с параметром *cbInBuffer*, указывающим, сколько байт передано.

Параметр *SIO\_GET\_QOS* используется после приема события *FD\_QOS*. Когда приложение получает уведомление об этом событии, для выяснения его причины следует вызвать *WSAIocctl* с параметром *SIO\_GET\_QOS*. Как уже упоминалось, событие *FD\_QOS* может быть сгенерировано из-за изменения пропускной способности сети или повторного согласования условий партнером. Чтобы получить значения QoS для сокета, передайте достаточно большой буфер как *ipvOutBuffer* с параметром *cbOutBuffer*, указывающим размер. Входными параметрами могут быть *NULL* и 0.

Единственная сложность при вызове *SIO\_GET\_QOS* — передать буфер, достаточно большой, чтобы вместить структуру *QOS*, включая объекты, относящиеся к поставщику. Поле *ProviderSpecific* представляет собой структуру *WASBUF* и находится внутри структуры *QOS*. Если поле *len* равно *QUERY\_PS\_SIZE*, а поле *buf*— 0, после возвращения из *WSAIocctl* поле *len* получит новое значение. Если вызов функции не удастся из-за слишком маленького буфера, поле *len* также получит новое правильное значение. Запрос точного размера буфера поддерживается только в Windows 2000. В Windows 98 просто предоставьте большой буфер.

С функцией *WSAIocctl* также можно использовать *ioctl*-команду *SIO\_CHK\_QOS* для запроса значений шести флагов:

- ***ALLGWTO SEND DATA*** — сообщает, может ли передача данных начаться немедленно или приложение должно ожидать сообщения RESV; возвращаемое значение — *BOOL*;
- ***ABLETORESVRSVP*** — сообщает отправителям, поддерживает ли их интерфейс протокол RSVP; возвращаемое значение — *BOOL*;



**III LINERATE** — возвращает сведения о пропускной способности интерфейса; возвращаемое значение — *DWORD*;

**III LOCAL TRAFFIC CONTROL** — сообщает, установлен ли и доступен ли для использования TC; возвращаемое значение — *BOOL*;

**Ж LOCALQoSABILITIES** — сообщает, доступна ли QoS; возвращаемое значение — *BOOL*;

**III ENDOENDQOSABILITY** — сообщает, доступна ли в сети сквозная QoS; возвращаемое значение — *BOOL*.

Когда вы вызываете команду *SIO\_CHK\_QOS*, параметр *ipInBuffer* указывает на значение типа *DWORD*, содержащее один из трех флагов. Параметр *ipOutBuffer* также должен ссылаться на *DWORD*, и по возвращении содержит запрошенное значение. Чаще всего используется флаг *ALLOWED\_TO\_SEND\_DATA*. Его применяют отправители, которые инициировали сообщение PATH, но не получили сообщение RESV, указывающее на успешное выделение уровня QoS. Когда отправители используют ioctl-команду *SIO\_CHK\_QOS* с флагом *ALLOWED\_TO\_SEND\_DATA*, у сети запрашивается, достаточно ли в настоящий момент скорость передачи негарантированного трафика для передачи данных, описанных в соответствующей структуре *QOS* (см. главу 9)–

Перечисленные выше флаги, которые возвращают значение *BOOL*, на самом деле возвращают 1 или 0, что соответственно означает «да» или «нет». Последние четыре флага могут возвращать константу *INF\_NOT\_AVAILABLE*, если в настоящий момент ответ получить не удалось.

## Завершение QoS

Завершение обработки RSVP и TC для сокета вызывает каждое из следующих Событий:

- закрытие сокета с помощью функции *closesocket*;
- III** завершение работы сокета с помощью функции *shutdown*;
- вызов функции *WSAConnect* с нулевым адресом партнера;
- III** вызов функции *WSALocctl* и *SIO\_SET\_QOS* с типами обслуживания *SERVICE\_TYPE\_NOTRAFFIC* или *SERVICE\_TYPE\_BESTEFFORT*.

Это вполне очевидно, кроме работы функции *shutdown*. Она может оповестить о прекращении передачи или приема, в результате чего будет завершен поток данных только для этого направления. Другими словами, если функция *shutdown* вызывается с параметром *SD\_SEND*, QoS по-прежнему будет влиять на получаемые данные.

## Объекты, относящиеся к поставщику

Объекты, рассматриваемые в этой главе, передаются как часть поля *Pg-providerSpecific* структуры *QOS*. Они либо возвращают информацию QoS приложению с помощью события *FD\_QOS*, либо их можно передать вместе с другими параметрами QoS вызову функции *WSALocctl* с параметром *SIO\_SET\_QOS*, чтобы перенастроить работу QoS.

Каждый объект, относящийся к поставщику, содержит в качестве первого элемента структуру *QOS\_OBJECT\_HDR*, которая определяет тип объекта. Это необходимо, поскольку объекты в основном возвращаются внутри структуры *QOS* после вызова *SIOjGET^QOS*. С помощью *QOSJDBJECTJ4DR* приложение может идентифицировать каждый объект и определить его важность. Заголовок объекта определен таю

```
typedef struct
{
    ULONG    ObjectType;
    ULONG    ObjectLength;
} QOS_OBJECT_HDR, *LPQOS_OBJECT_HDR;
```

*ObjectType* определяет тип предустановленного объекта, *ObjectLength* сообщает о его длине, включая заголовки. Тип объекта может быть одним из флагов, перечисленных в табл. 12-1.

Табл. 12-1. Типы объектов

Объект, относящийся к поставщику	Структура объекта	
<i>QOS_OBJECT_PRIORITY</i>	<i>QOS_PRIORITY</i>	38. ViO
<i>QOSOBJECT_SD_MODE</i>	<i>QOSSDjMODE</i>	.пой
<i>QOSOBJECT_TRAFFIC_CIASS</i>	<i>QOS_TRAFFIC_CLASS</i>	^ „,
<i>QOS_OBJECT_DESTADDR</i>	<i>QOSJDESTADDR</i>	
<i>QOS_OBJECT_SHAPER_QUEUE_DROP^MODE</i>	<i>QOS_SHAPER_QUEUEJMIT_DROP_MO</i>	
<i>QOSJBJECT_SHAPER_QUEUEJMIT</i>	<i>QOS_SHAPER_QUEUEJMIT</i>	
<i>RSVP_OBJECT_STATUSJNFO</i>	<i>RSVPSTATUSJNFO</i>	
<i>RSVPJDBJECTRESERVEJNFO</i>	<i>RSVP_RESERVE_INFO</i>	
<i>RSVPJDBJECT_ADSPEC</i>	<i>RSVPADSPEC</i>	
<i>RSVPJDBJECTPOLICYJNFO</i>	<i>RSVPJPOLICYJNFO</i>	
<i>QOS_OBJECT_END_OFJIST</i>	Отсутствует. Объектов больше нет.	

## Приоритет QoS

Приоритет QoS определяет абсолютный приоритет потока. Уровни приоритета лежат в диапазоне от 1 до 7, от низшего к высшему. Структура *QOSPRIORITYTY* определена таю

```
typedef struct _QOS_PRIORITY
{
    QOSOBJECTHDR ObjectHdr;
    UCHAR          SendPriority
    UCHAR          SendFlags;
    UCHAR          ReceivePriority
    UCHAR          Unused;
    QOS_PRIORITY, *LPQOS_PRIORITY;
```

Эти значения определяют локальный приоритет (внутренний для отправляющего узла) соответствующего трафика потока относительно трафика из других потоков. По умолчанию приоритет потока — 3- Этот приоритет ис-

пользуется в сочетании с параметром *ServiceType* структуры *FLOWSPEC* для определения приоритета, который должен быть применен к потоку, внутреннему для планировщика пакетов. Поля *SendFlags* и *ReceivePriority* в настоящее время не используются, но возможно, будут применяться в будущем.

### Режим отбрасывания данных

Объект *QOS* определяет, как элемент Packet Shaper (формировщик пакетов) модуля TC обрабатывает данные определенного потока. Это свойство обычно используется, когда потоки не соответствуют параметрам, заданным в структуре *FLOWSPEC*, то есть если приложение передает данные со скоростью, превышающей заданную в поле *TokenRate* отправляющей структуры *FLOWSPEC*. Объект определяет, как локальная система действует в этом случае. Вот описание структуры *QOS\_SD\_MODE*:

```
typedef struct _QOS_SD_MODE
{
    QOS_OBJECT_HDR   ObjectHdr,
    ULONG             ShapeDiscardMode,
} QOS_SD_MODE, *PQOS_SD_MODE;
```

Поле *ShapeDiscardMode* может принимать одно из следующих значений. И ***TCNONCONFBORROW*** — поток получает ресурсы, оставшиеся после обработки всех потоков с более высоким приоритетом. Потоки этого типа не подчиняются ни формировщику (Shaper), ни секвенсору (Sequencer). Если задано значение для *TokenRate*, пакеты будут признаны не соответствующими условиям передачи, а их приоритет — снижен до приоритета, более низкого, чем для негарантированного трафика.

**III *TCNONCONFSHAPE*** — должно быть задано значение для *TokenRate*. Не соответствующие пакеты будут храниться в формировщике до тех пор, пока не станут соответствовать условиям передачи.

**K *TCNONCONFDISCARD*** — должно быть задано значение для *TokenRate*. Не соответствующие пакеты будут отброшены.

Зачем применять режим ***TC\_NONCONFDISCARD***, если при этом данные могут быть отброшены еще до начала передачи? Это может понадобиться, например, при передаче звуковых или видеоданных. В большинстве случаев структура *FLOWSPEC* настраивается для передачи пакета, размер которого равен одному кадру видео- или небольшому фрагменту звуковых данных. Если по какой-либо причине пакет не соответствует условиям, нужно ли ждать, пока он будет признан годным к передаче (как в случае с ***TC\_NONCONFSHAPE***)? Не лучше ли отбросить этот пакет и перейти к следующему? Для критичных по времени данных, таких как видеоданные, следует именно так и поступить.

&lt;

### Класс трафика QoS

Структура *QOS\_TRAFFIC\_CLASS* может передавать параметр класса трафика 802.1p, поставляемый узлу сетевым устройством канального уровня.

```
typedef struct _QOS_TRAFFIC_CLASS
{
    QOS_OBJECT_HDR    ObjectHdr,
    ULONG              TrafficClass,
} QOS_TRAFFIC_CLASS *LPQOS_TRAFFIC_CLASS
```

Узлы маркируют MAC-заголовки соответствующих передаваемых пакетов значением *TrafficClass*, заданным в структуре. Хотя эта структура включена в *Qos h*, приложения не могут сами задавать приоритет.

## Адрес места назначения QoS

Структура *QOSDESTADDR* используется для задания адреса места назначения для отправляющего сокета, не требующего соединения, без вызова функции *WSAConnect*. Никакие RSVP-сообщения PATH и RESV не будут отправлены, пока не выяснится адрес места назначения не требующего соединения сокета. Этот адрес можно задать с помощью ioctl-команды *SIO\_SET\_QOS*.

```
typedef struct _QOS_DESTADDR
{
    QOS_OBJECT_HDR    ObjectHdr,
    const struct sockaddr *SocketAddress,
    ULONG              SocketAddressLength,
} QOS_DESTADDR, *LPQOS_DESTADDR, 4QVJ
```

Поле *SocketAddress* ссылается на структуру *SOCKADDR* задающую адрес конечной точки для определенного протокола. *SocketAddressLength* — это размер структуры *SOCKADDR*.

## Режим отбрасывания данных при переполнении очереди формировщика

Структура *QOS\_SHAPER\_QUEUEJMIT\_DROP\_MODE* задает замещение стандартной схемы отбрасывания пакетов, если достигнут лимит очереди формировщика для потока. Формировщик трафика (Traffic Shaper) модуля ТС можно настроить так, чтобы он отбрасывал любые избыточные данные, если находящиеся в настоящий момент в очереди данные не соответствуют структуре *FLOWSPEC*. Структура *QOSJHAPER\_QUEUEJMIT\_DROP\_MODE* определена так:

```
typedef struct _QOS_SHAPER_QUEUE LIMIT_DROP_MODE
{
    QOS_OBJECT_HDR    ObjectHdr
    ULONG              DropMode,
> QOS_SHAPER_QUEUE_LIMIT_DROP_MODE,
  *LPQOS_SHAPER_QUEUE_LIMIT_DROP_MODE,
```

Возможны два значения *DropMode*:

- III **QOSSHAPERDROPFROMHEAD** — отбрасывает пакеты из начала очереди (действие по умолчанию),
- III **QOSSHAPERDROPINCOMING** — отбрасывает любые входящие пакеты по достижении лимита очереди

### Лимит очереди формировщика QoS

Структура *QOS\_SHAPER\_QUEUE\_LIMIT* позволяет изменить стандартный для потока лимит очереди формировщика

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT
{
    QOS_OBJECT_HDR    ObjectHdr,
    ULONG              QueueSizeLimit,
} QOS_SHAPER_QUEUE_LIMIT, *LPQOS_SHAPER_QUEUE_LIMIT,
```

*QueueSizeLimit* — это размер очереди формирователя в байтах. Большой размер очереди формировщика предотвратит потерю данных из-за\* недостатка места в буфере.

### Информация о состоянии RSVP

Объект *RSVP\_STATUS\_INFO* используется для возвращения информации о состоянии и характерных ошибках RSVP

```
typedef struct _RSVP_STATUS_INFO {
    QOS_OBJECT_HDR    ObjectHdr,
    ULONG              StatusCode
    ULONG              ExtendedStatus1,
    ULONG              ExtendedStatus2,
} RSVP_STATUS_INFO, *LPRSV_P_STATUS_INFO,
```

Поле *StatusCode* — это возвращенное сообщение RSVP. Возможны следующие коды:

**III WSAQOS\_RECEIVERS** — поступило хотя бы одно сообщение RSVP,

- **WSAQOSSENDERS** — поступило хотя бы одно сообщение PATH,
- **WSA\_NO\_QOS\_RECEIVERS** — отсутствуют приемники,
- **WSANOQOSSENDERS** — отсутствуют отправители,
- **WSAQOSREQUESTCONFIRMED** — резервирование подтверждено,
- **WSAQOSADMISSIONFAILURE** — запрос отклонен из-за недостатка ресурсов,
- **WSAQOSPOLICYFAILURE** — запрос отклонен из-за административных причин или неверных учетных сведений,
- **WSAQOSBADSTYLE** — неизвестный или конфликтующий стиль,
- **WSAQOSBADOBJECT** — возникла проблема с какой-либо частью структуры *RSVPFILTERSPEC* или буфером поставщика,

**III WSAQOSTRAFFICCTRLERROR** — возникла проблема с какой-либо частью структуры *FLOWSPEC*,

**III WSAQOSGENERICERROR** — общая ошибка,

**III ERRORIOPENDING** — перекрытая операция отменена

Другие два поля — *ExtendedStatus1* и *ExtendedStatus2*, зарезервированы для информации о поставщике.

Обычно после получения сообщения RSVP приложение получает событие *FD\_QOS* и вызывает *SIO\_GET\_QOS* для приема структуры *QOS*, содержащей объект *RV\_STATUS\_INFO*. Например, для QoS-совместимого UDP-приемника событие *FD\_QOS*, содержащее сообщение *WSA\_QOS\_SENDERS*, генерируется, чтобы сообщить, что кто-то запросил службу QoS для передачи данных приемнику.

### Информация о резервировании RSVP

Объект *RSVPJRESERVEINFO* хранит RSVP-информацию для тонкой настройки QoS средствами Winsock 2. QoS API и буфер со сведениями о поставщике. Объект *RSVP\_RESERVE\_INFO* замещает стандартный стиль резервирования и используется приемником QoS.

```
typedef struct _RSVP_RESERVE_INFO
{
    QOSOBJECT_HDR    ObjectHdr,
    ULONG            Style,
    ULONG            ConfirmRequest,
    ULONG            NumPolicyElements,
    LPRSV.POLICY      PolicyElementList,
    ULONG            NumFlowDesc,
    LPFLOWDESCRIPTOR FlowDescList,
} RSVP_RESERVE_INFO, *LRSVP_RESERVE_INFO,
```

Поле *Style* задает тип фильтра, применяемого к этому приемнику. В табл. 12-2 приведены доступные типы фильтров и типы фильтров по умолчанию, используемые разными типами приемников.

**Табл. 12-2. Стили фильтров по умолчанию**

Стиль фильтра	Пользователи по умолчанию
Фиксированный фильтр	Одноадресные приемники UDP приемники (с установлением соединения)
Фильтр, содержащий метасимволы	Многоадресные приемники UDP-приемники (без установления соединения)
Общий явный фильтр	Отсутствуют

Если значение поля *ConfirmRequest* отлично от 0, после приема запроса RSVP принимающим приложениям будет отправлено уведомление. Поле *NumPolicyElements* связано с полем *PolicyElementList*. Оно содержит количество объектов *RSVP\_POLICY*, хранимых в поле *PolicyElementList* (объект *RSVP\_POLICY* будет описан далее в этой главе).

Рассмотрим разные стили фильтров и характеристики каждого из них более подробно.

**Флаг *RSVPDEFAULTSTYLE*** сообщает поставщику службы QoS о необходимости использовать стиль по умолчанию. В табл. 12-2 приведены стили по умолчанию для разных приемников. Одноадресные приемники ис-

пользуют фиксированный фильтр, в то время как многоадресные — фильтр, содержащий метасимволы. UDP-приемники, вызывающие *WSAConnect*, также используют фиксированный фильтр

**Стиль *RSVPFIXEDFILTERSTYLE*** обычно устанавливает один поток с гарантиями QoS между приемником и единственным источником. Для одноадресного приемника и подключенных UDP-приемников *NumFlowDesc* имеет значение 1, а *FlowDescList* — содержит адрес отправителя. Между тем, можно задать стиль с несколькими фиксированными фильтрами, позволяющий приемнику резервировать взаимоисключающие потоки от нескольких явно заданных источников. Например, если приемник будет получать данные от трех отправителей и требует гарантированную пропускную способность 20 кбит/с для каждого, используйте стиль нескольких фиксированных фильтров. Тогда *NumFlowDesc* будет иметь значение 3, а *FlowDescList* — содержать три адреса — по одному для каждой структуры *FLOWSPEC*.

Можно также присваивать каждому отправителю разные уровни QoS, но они не должны быть одинаковыми. Заметьте: одноадресные приемники и подключенные UDP-приемники не могут использовать несколько фиксированных фильтров. На рис. 12-1 показана связь между структурами *FLOWDESC* и *RIPTORnRSVP FILTERSPEC*.

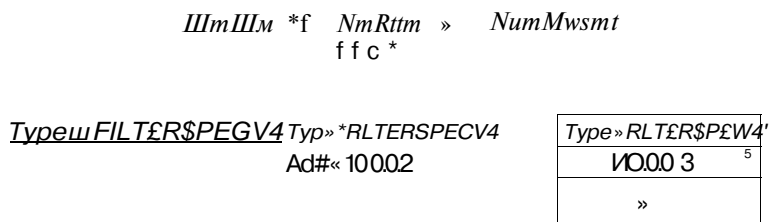


Рис. 12-1. Стиль с несколькими фиксированными фильтрами

**Стиль *RSVPJWILDCAKDSTYLE***. Многоадресные и неподключенные UDP-приемники используют фильтр, содержащий метасимволы. Для использования этого стиля в случае соединений TCP или для подключенных UDP-приемников присвойте полю *NumFlowDesc* значение 0, а полю *FlowDescList* — *NULL*. Это стандартный стиль фильтра для неподключенных UDP-приемников и приложений многоадресного вещания, поскольку адрес отправителя неизвестен.

**Стиль *RSVPSHAREEXPLICITSTYLE*** похож на стиль с несколькими фиксированными фильтрами, за исключением того, что сетевые ресурсы распределяются между всеми отправителями, а не выделяются для каждого. В этом случае поле *NumFlowDesc* равно 1, а поле *FlowDescList* содержит список адресов отправителей (рис. 12-2).

*ObjectHdr*

*Style* *RSVP* *SHAREDEXPLICIT* *JT* *IL*

*ConfirmRequest*

*if* *um* *Pohcy* *Eiements*

*policyElementList*

*NumFlowDesc* = 1

*NumFilterList* = 3

• *FlowDescList*

*FilterList*

F

<i>Type</i> «ЯИТЕH\$PIИ4	<i>Type</i> * <i>Fi</i> <i>LTERSPEC</i> <i>W</i>	<i>Tm</i> « <i>HLTERSPEC</i> <i>V</i> <i>M</i>
<i>Addr</i> * <i>AMQA</i>	<i>Addr</i> « <i>IO.Q.02</i>	<i>Addr</i> « <i>IO.ИДЗ</i> <i>I</i>
		<i>i</i>

Рис. 12-2. Общий явный фильтр

Последние два поля — *NumFlowDesc* и *FlowDescList*, рассматривались при обсуждении стилей *RSVP*. Использование этих полей зависит от стиля. *NumFlowDesc* задает количество объектов *FLOWDESCRIPTOR* в поле *FlowDescList*:

`typedef struct FLOWDESCRIPTOR`

шц

```

    FLOWSPEC          FlowSpec;
    ULONG              NumFilters;
    LPRSVF_FILTERSPEC FilterList;
} FLOWDESCRIPTOR, «LPFLOWDESCRIPTOR;
    {

```

Этот объект используется для определения типов фильтров для каждой структуры *FLOWSPEC*, передаваемых *FlowSpec*. Поле *NumFilters* содержит количество объектов *RSVP\_FILTERSPEC*, имеющихся в массиве *FilterList*. Объект *RSVP\_FILTERSPEC* определен так:

```

typedef struct _RSVP_FILTERSPEC <
    FilterType  Type;
    union {
        RSVP_FILTERSPEC_V4  FilterSpecV4;
        RSVP_FILTERSPEC_V6  FilterSpecV6;
        RSVP_FILTERSPEC_V6_FLOW  FilterSpecV6Flow;
        RSVP_FILTERSPEC_V4_GPI  FilterSpecV4Gpi;
        RSVP_FILTERSPEC_V6_GPI  FilterSpecV6Gpi;
    };
} RSVP_FILTERSPEC, •LPRSVF_FILTERSPEC;

```

Поле *Type* — это простое перечисление следующих значений:

```

typedef enum {
    FILTERSPECV4 = 1,
    FILTERSPECV6,
    FILTERSPECV6_FLOW,
    FILTERSPECV4_GPI,
    FILTERSPECV6_GPI,
    FILTERSPEC_END
} FilterType;

```



Это перечисление задает объект, имеющийся в объединении. Вот эти спецификации фильтров:

```
typedef struct _RSVP_FILTERSPEC_V4 {
    IN_ADDR_IPV4    Address;
    USHORT          Unused;
    USHORT          Port;
} RSVP_FILTERSPEC_V4, *LPRSVF_FILTERSPEC_V4;

typedef struct _RSVP_FILTERSPEC_V6 {
    IN_ADDR_IPV6    Address;
    USHORT          Unused;
    USHORT          Port;
} RSVP_FILTERSPEC_V6, *LPRSVF_FILTERSPEC_V6;

typedef struct _RSVP_FILTERSPEC_V6_FLOW {
    IN_ADDR_IPV6    Address;
    UCHAR          Unused;
    UCHAR          FlowLabel[3];
} RSVP_FILTERSPEC_V6_FLOW, *LPRSVF_FILTERSPEC_V6_FLOW;

typedef struct _RSVP_FILTERSPEC_V4_GPI {
    IN_ADDR_IPV4    Address;
    ULONG          GeneralPortId;
} RSVP_FILTERSPEC_V4_GPI, *LPRSVF_FILTERSPEC_V4_GPI;

typedef struct _RSVP_FILTERSPEC_V6_GPI {
    IN_ADDR_IPV6    Address;
    ULONG          GeneralPortId;
} RSVP_FILTERSPEC_V6_GPI, *LPRSVF_FILTERSPEC_V6_GPI;
```

## Объект *RSVP\_ADSPEC*

Объект *RSVP\_ADSPEC* задает информацию, передаваемую в RSVP Adspec. Этот RSVP-объект обычно указывает, какие типы служб доступны (путем управляемой загрузки или гарантированно), столкнулось ли сообщение PATH с не-RSVP переходом, а также минимальный MTU по пути передачи. Его структура определена так:

```
typedef struct _RSVP_ADSPEC {
    QOS_OBJECT_HDR    ObjectHdr;
    AD_GENERAL_PARAMS GeneralParams;
    ULONG             NumberOfServices;
    CONTROL_SERVICE    Services[1];
} RSVP_ADSPEC, *LPRSVF_ADSPEC;
```

Поле *GeneralParams* — структура типа *AD\_GENERAL\_PARAMS*, задающая некоторые общие параметры:

```
typedef struct _AD_GENERAL_PARAMS
```

```

ULONG      IntServAwareHopCount;
ULONG      PathBandwidthEstimate
ULONG      MinimumLatency;
ULONG      PathMTU;
ULONG      Flags;
} AD_GENERAL_PARAMS, *LPAD_GENERAL_PARAMS;

```

*IntServAwareHopCount* — это количество переходов, соответствующих требованиям службы Integrated Services (IntServ). *PathBandwidthEstimate* — минимальная пропускная способность, доступная от отправителя до приемника. *MinimumLatency* — сумма минимальных задержек (в микросекундах) процессов передачи пакетов в маршрутизаторах. *PathMTU* — максимальная единица сквозной передачи, которая не подвергается фрагментации. Поле *Flags* в настоящее время не используется.

### Сведения о политике RSVP

Это последний из объектов, относящихся к поставщику, который мы рассмотрим. Он не очень понятен, так как содержит произвольное количество элементов политики RSVP, которые не определены. Структура *RSVPPOLICYINFO* определена так:

```

typedef struct _RSVP_POLICY_INFO {
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG             NumPolicyElement;
    RSVP_POLICY        PolicyElement[1];
} RSVP_POLICY_INFO, *LPRSVP_POLICY_INFO;

```

Поле *NumPolicyElement* содержит информацию о количестве структур *RSVP POLICY*, имеющих в массиве *PolicyElement*.

```

typedef struct _RSVP_POLICY {
    USHORT    Len;
    USHORT    Type;
    UCHAR     Info[4];
} RSVP_POLICY, *LPRSVP_POLICY;

```

Структура *RSVP\_POLICY* — это данные, передаваемые RSVP от имени компонента политики и не имеющие прямого отношения к обсуждаемым темам.

## Программирование QoS

Инициирование сеанса RSVP — основная часть QoS. Пропускная способность не резервируется для процесса, до тех пор пока RSVP-сообщения PATH и RESV не будут отправлены и обработаны. Приложениям важно знать, когда генерируются RSVP-сообщения. До генерирования сообщения PATH отправителю должны быть известны отправляющий элемент *FLOWSPEC*, IP-адрес и порт источника, целевой IP-адрес, порт и протокол.

Элемент *FLOWSPEC* становится известен всякий раз, когда вызывается поддерживающая QoS функция: например, *WSAConnect*, *WSAJoinLeaf*, *WSAIoctl* (с параметром *SIO\_SET\_QOS*) и т. д. Исходный IP-адрес и порт неизвестны. Пока сокет связан локально: неявно (например, в рамках соединения) или

явно. Наконец, приложению требуется место назначения данных. Необходимая информация собирается либо после вызова соединения, либо в случаях, не требующих соединений UDP. Для этого следует задать объект *QOS\_DESTADDR* в данных, относящихся к поставщику (передаются с помощью ioctl-команды *SIO\_SET\_QOS*).

Для генерации RSVP-сообщения RESV также необходимо знать принимающий элемент структуры *FLOWSPEC*, адрес и порт каждого отправителя, локальный адрес и порт принимающего сокета.

Принимающий элемент *FLOWSPEC* получают из любой поддерживающей QoS функции Winsock. Адрес и порт каждого отправителя зависят от стиля фильтра, который задают вручную через относящуюся к поставщику *структуру RSVP RESERVE INFO*. Иначе эта информация может быть получена из сообщения PATH. Разумеется, не всегда необходимо сообщение PATH, чтобы получить адрес отправителя для генерации сообщений RESV — это зависит от типа сокета. Пример —используемый при многоадресной рассылке фильтр, содержащий метасимволы. Отправленное сообщение RESV применяется ко всем отправителям в сеансе. Объяснение локального адреса и порта очевидно для одноадресных и UDP-приемников, но не для многоадресных. В случае многоадресных приемников локальный адрес и порт — это адрес многоадресной рассылки и соответствующий номер его порта.

В этом разделе мы прежде всего рассмотрим различные типы сокетов и их взаимодействие с поставщиком службы QoS и RSVP-сообщениями. Затем ознакомимся с тем, как поставщик QoS сообщает приложениям об определенных событиях. Знать эти концепции, способы получения гарантий QoS, а также условия действия и изменения этих гарантий необходимо для успешного написания QoS-совместимых приложений.

## RSVP и типы сокетов

Мы рассмотрим следующие типы сокетов: UDP, TCP и многоадресный UDP, а также их взаимодействие с поставщиком службы QoS для генерации сообщений PATH и RESV.

### Одноадресный UDP

Поскольку можно использовать и подключенные, и неподключенные сокеты UDP, чтобы включить QoS на одноадресных сокетах UDP, нужно определить не так уж много параметров. В случае отправителя UDP отправляющая структура *FLOWSPEC* может быть получена от одной из вызывающих QoS функций. Локальный адрес и порт получают либо в результате явной привязки, либо в ходе неявной привязки средствами *WSAConnect*. Последняя часть — это адрес и порт принимающего приложения, которые могут быть определены либо в *WSAConnect*, либо посредством относящейся к поставщику структуры *QOS\_DESTADDR*, передаваемой через параметр *SIO\_SET\_QOS*. Если для включения QoS используется *SIO\_SET\_QOS*, сокет должен быть привязан заранее.

Для UDP-приемника функцию *WSAConnect* вызывают, чтобы ограничить принимающее приложение одним отправителем. Кроме того, приложения мо-

гут задавать структуру *QOSIDESTADDR* с помощью *ioctl*-команды *SIO\_SET\_QOS*. Иначе функция *SIO\_SET\_QOS* может быть вызвана без предоставления адреса места назначения. В этом случае сообщение RESV будет сгенерировано с помощью фильтра, содержащего метасимволы. Задавать адрес места назначения посредством функции *WSAConnect* или структуры *QOSDESTADDR* следует, только если вы хотите, чтобы приложение получало данные лишь от одного отправителя, использующего стиль с фиксированным фильтром.

UDP-приемник может вызвать функцию *WSAConnect* и *ioctl*-команду *SIO\_SET\_QOS* в любом порядке. Если первой вызвана *SIO\_SET\_QOS*, то сообщение RESV сначала создается с фильтром, содержащим метасимволы. После вызова соединения предыдущий сеанс RESV прерывается, и создается новый сеанс со стилем фиксированного фильтра. Если же *SIO\_SET\_QOS* вызывается после *WSAConnect* и фиксированного фильтра, сообщение RSVP не прерывает сеанс RSVP и не генерирует стиль фильтра, содержащего метасимволы. Вместо этого оно просто обновляет параметры QoS, связанные с имеющимся сеансом RSVP.

### Одноадресный TCP

Сеансы TCP можно подразделить на два типа. Отправителем может быть клиент, подключающийся к сети и отправляющий данные. В другом случае отправителем является сервер, к которому подключается клиент. Если отправитель — клиент, параметры QoS вы вправе задать прямо в вызове *WSAConnect*, в результате будет отправлено сообщение PATH. *ioctl*-команду *SIO\_SET\_QOS* также можно вызвать до вызова соединения, но сообщения PATH не будут сгенерированы, пока один из вызовов соединения не будет знать целевой адрес.

Если отправитель — сервер, он вызывает функцию *WSAAccept*, чтобы установить соединение с клиентом. Эта функция не позволяет задавать QoS на принятом сокете. Если QoS задан до вызова *WSAAccept* с помощью *SIO\_SET\_QOS*, любой принятый сокет наследует уровни QoS, заданные на прослушивающем сокете. Вообще-то, если отправитель использует условную функцию в *WSAAccept*, функция должна передать значения QoS, заданные на подключающемся клиенте. Но это не так. И в Windows 98, и в Windows 2000 поставщик службы QoS передает блок данных. Исключение, — если в Windows 98 параметр *ipSQOS* не равен 0, какие-то значения QoS должны быть заданы с помощью *ioctl*-команды *SIO\_SET\_QOS* внутри условной функции. Иначе вызов *WSAAccept* не будет успешен, даже если возвратится значение *CFACCEPT*. QoS можно также задать на клиентском сокете, после того как он принят.

Рассмотрим принимающие приложения TCP. Первый вариант — вызов функции *WSAConnect* с принимающей структурой *FLOWSPEC*. Когда это происходит, поставщик службы QoS создает запрос RESV. Если параметры QoS не передаются *WSAConnect*, *ioctl*-команду *SIO\_SET\_QOS* задают позднее (в результате чего отправляется сообщение RESV). Последнее сочетание — это сервер, являющийся приемником, что аналогично случаю передачи. QoS можно включить на прослушивающем сокете до вызова *WSAAccept*, при этом клиентский сокет наследует те же самые уровни QoS. Иначе QoS включается в условной функции или после приема сокета. В любом случае поставщик службы QoS генерирует сообщение RESV, как только поступает сообщение PATH.

## Многоадресная рассылка

Многоадресные отправители действуют так же, как отправители UDP, но функция *WSAJoinLeaf* используется, чтобы стать членом многоадресной группы (в противоположность вызову *WSAConnect* с адресом места назначения). QoS можно включить с помощью *WSAJoinLeaf* отдельно, посредством вызова *SIO\_SET\_QOS*. Адрес сеанса многоадресной рассылки используется для формирования объекта сеанса RSVP, включенного в RSVP-сообщение PATH.

В случае многоадресного приемника сообщения RSVP не будут генерироваться, пока не определен адрес многоадресной рассылки с помощью функции *WSAJoinLeaf*. Поскольку многоадресный приемник не задает адрес партнера, поставщик службы QoS генерирует сообщения RESV со стилем фильтра, содержащего метасимволы. Поставщик службы QoS не запрещает сокету присоединяться к нескольким многоадресным группам. В этом случае поставщик отправляет сообщения RESV для всех групп, имеющих соответствующее сообщение PATH. Параметры QoS, передаваемые каждой функцией *WSAJoinLeaf* используются в каждом сообщении RESV. Но если функция *SIO\_SET\_QOS* вызвана на соquete после присоединения к нескольким группам, новые параметры QoS применяются ко всем многоадресным группам, к которым присоединился сокет.

Когда данные отправляются многоадресной группе, QoS применяется только к данным, отправляющимся группе, к которой присоединился отправитель. Другими словами, если вы присоединяетесь к одной многоадресной группе и используете функцию *sendto/WSASendTo*, имеющую любую другую многоадресную группу в качестве места назначения, QoS к этим данным не применяется. Кроме того, если сокет присоединяется к многоадресной группе, задающей определенное направление (например, *SENDERJDONLY* или *JL\_RECEIVERJDONLY* в параметре *dwFlags*, передаваемом функции *WSAJoinLeaf*), QoS применяется соответственно. Сокет, настроенный только для приема, не получит никаких преимуществ QoS для отправленных данных.

## Уведомления QoS

Мы рассказали, как вызвать QoS для сокетов TCP, UDP, многоадресного UDP, о соответствующих RSVP-событиях, происходящих в зависимости от того, отправляете вы данные или получаете. Между тем, завершение этих сообщений RSVP не строго привязано к вызовам API, которые их инициируют. Вызов *WSAConnect* для принимающего TCP-сокета генерирует сообщение RESV, но это сообщение не зависит от вызова API: вызов возвращается, не подтверждая резервирования и выделения сетевых ресурсов. По этой причине было добавлено новое асинхронное событие *FD\_QOS*, передаваемое сокету.

Как правило, уведомление о событии *ED\_QOS* передается, если имеется:

- уведомление о принятии или отклонении запроса QoS от приложения;
- значительные изменения в QoS, переданные сетью (не совместимые с согласованными значениями);

**III** состояние, определяющее, готов ли партнер QoS передавать или принимать данные для определенного потока.

## Регистрация для получения уведомлений *FD\_QOS*

Чтобы получать уведомления о событиях *FD\_QOS*, приложение должно зарегистрироваться. Для этого существует несколько способов. Прежде всего можно воспользоваться функцией *WSAEventSelect* или *WSAAsyncSelect* и включить флаг *FD\_QOS* в побитовую операцию ИЛИ над флагами событий. Однако приложение готово продолжать прием события *FD\_QOS*, только если уже был сделан вызов одной из инициирующих QoS функций. Заметьте: в некоторых случаях приложению может потребоваться получить событие *FD\_QOS* без задания уровней QoS на сожете. Этого можно достичь, задав структуру *goS*, отправляющие и принимающие элементы *FLOWSPEC* которой содержат флаг *QOS\_NOT\_SPECIFIED* или *SERVICETYPE\_NOTRAFFIC*. При этом над флагами *SERVICE\_NO\_QOS\_SIGNALING* и *SERVICETYPE\_NOTRAFFIC* необходимо выполнить логическую операцию ИЛИ для направления QoS, в которой вы хотите получить уведомление о событии.

Подробнее о вызовах двух асинхронных функций выбора — в главе 8. Если вы используете *WSAEventSelect* после того, как событие произошло, вызовите функцию *WSAEnumNetworkEvents*, чтобы получить дополнительные доступные коды состояния. Эта функция также описана в главе 8, но мы рассмотрим ее и в этой главе, так как она важна для приложений QoS. Передайте описатель сокета, описатель события и объект *WSANETWORKEVENTS* вызову функции, который вернет информацию о событии этой структуре.

```
typedef struct WSANETWORKEVENTS
{
    long    lNetworkEvents;
    int     iErrorCode[FO_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

Поле *lNetworkEvents* будет хранить результат логической операции ИЛИ для всех сработавших флагов события. Чтобы обнаружить, что событие произошло, просто выполните логическую операцию И (оператором &) над этим полем и флагом события. Если результат отличен от 0, событие произошло. Массив *iErrorCode* сообщит об ошибках или, в случае QoS, проинформирует о состоянии.

Если событие произошло, относящийся к нему флаг используется для указания позиции в этом массиве. Если индекс массива равен 0, ошибки не было, иначе индекс укажет на элемент массива, содержащий код ошибки. Например, если событие *FD\_QOS* произошло, используйте флаг *FD\_QOS\_BIT* для указания на массив *iErrorCode*, чтобы проверить любые ошибки или получить информацию о состоянии. Все другие асинхронные события (*FD\_READJ3IT*, *FD\_WRITEJ3IT* и т. п.) имеют похожие флаги указателей.

## Уведомления RSVP

Существуют два способа приема уведомлений QoS. Оба они тесно связаны с темой этого раздела, то есть получением результатов события QoS. Если вы зарегистрировались для получения уведомлений *FD\_QOS* посредством *WSAAsyncSelect* или *WSAEventSelect* и действительно получаете уведомление о собы-

тии *FD\_QOS*, вызовите *WSAIoctl* с *ioctl*-параметром *SIO\_GET\_QOS*, чтобы выяснить, чем вызвано событие. На самом деле, регистрироваться для получения событий *FD\_QOS* не обязательно — можно просто вызвать *WSAIoctl* с командой *SIO\_GET\_QOS*, воспользовавшись перекрытым вводом-выводом. Для этого определите порядок завершения, вызываемый, когда поставщик службы QoS обнаруживает изменение в QoS. Когда произойдет обратный вызов, структура QoS будет доступна в выходном буфере.

В любом случае, если в QoS произошло изменение, ваше приложение будет уведомлено о нем посредством регистрации для *FD\_QOS* или путем использования перекрытого ввода-вывода и *SIO\_GET\_QOS*. Если вы регистрируетесь для *FD\_QOS*, то при уведомлении о событии нужно также вызывать *WSAIoctl* с *ioctl*-командой *SIO\_GET\_QOS*. В обоих случаях возвращаемая структура *QOS* содержит информацию только по одному направлению. Другими словами, полю *ServiceType* структуры *FLOWSPEC* для неверного направления должно быть присвоено значение *SERVICETYPE\_NOCHANGE*. Если произошло несколько событий QoS, вызывайте *WSAIoctl* и *SIO\_GET\_QOS* в цикле, пока не будет возвращено значение *SOCKET\_ERROR*, и *WSAGetLastError* не вернет значение *WSAEWOULDBLOCK*.

Последняя тонкость при вызове *SIO\_GET\_QOS* — размер буфера. Когда вызывается событие *FD\_QOS*, возможно, будут возвращены объекты, относящиеся к поставщику. Но чаще, если буфер достаточно велик, все же возвратится структура *RSVP\_STATUS\_INFO*. О выборе размера буфера мы рассказали ранее, при рассмотрении *WSAIoctl*.

Если приложение использует одну из функций асинхронных событий, помните, когда произошло событие *FD\_QOS*, всегда следует выполнить операцию *SIO\_GET\_QOS*, чтобы заново активизировать уведомления *FD\_QOS*.

Теперь обсудим существующие типы уведомлений. Первая и наиболее очевидная причина для события QoS — это изменение параметров *FLOWSPEC* для определенного потока. Например, если вы настраиваете сокет с негарантированным обслуживанием, поставщик службы QoS будет периодически отправлять уведомление приложению, информируя его о текущем состоянии сети. Кроме того, если вы задаете контролируемую нагрузку и т. п., после резервирования QoS-параметры размера корзины и скорости поступления блоков данных могут немного отличаться от тех, которые вы запросили. После приема уведомления QoS приложение должно сравнить возвращенное значение *FLOWSPEC* с запрошенным и убедиться, что может продолжать работу.

В течение всего времени существования QoS-совместимого сокета вы можете выполнить *SIO\_SET\_QOS*, чтобы изменить любые параметры, в результате чего будет сгенерировано уведомление партнеру или партнерам, связанным с текущим сеансом RSVP. Надежное приложение должно уметь обрабатывать эти ситуации.

Кроме обновления параметров QoS уведомление о событии QoS оповещает также о других событиях, таких как уведомления об отправителях или приемниках. Коды возможных событий приведены в разделе <Информация о состоянии RSVP>. Получить эти коды можно, во-первых, как часть объекта

*RSVP\_STATUSINFO* Когда происходит событие QoS и вызывается *SIO\_GET\_QOS*, объект *RSVP\_STATUSINFO* будет возвращен как часть буфера, относящегося к поставщику

Второй способ — если вы используете *WSAEventSelect*, чтобы зарегистрироваться для получения событий, эти коды вернутся в структуре *WSANETWORKEVENTS*, возвращаемой из *WSAEnumNetworkEvents*. Коды также можно найти в массиве *ErrorCode*, проиндексированном по полю *FD\_QOSJ3IT*. Первые пять из перечисленных кодов не являются кодами ошибок. Они возвращают важную информацию о состоянии соединения QoS. Остальные коды состояния не мешают передавать или принимать данные — они просто указывают на ошибку в сеансе QoS. Очевидно, что отправленные в такой ситуации данные не будут соответствовать запрошенным гарантиям QoS.

**Уведомления *WSAQOS\_RECEIVERS* и *WSAQOS\_NO\_RECEIVERS*.** При одноадресной рассылке после того, как отправитель начинает работу и получает первое сообщение RSVP, приложению передается *WSA\_QOS\_RECEIVERS*. Если приемник выполняет какие-либо действия по отключению QoS, генерируется сообщение RESV. После того, как отправитель получит это сообщение, приложению передается *WSA\_QOS\_NO\_RECEIVERS*. Очевидно, что при одноадресной рассылке многие приемники закрывают сокет, генерируя события *FD\_CLOSE* и *WSA\_QOS\_NO\_RECEIVERS*. В большинстве случаев в ответ приложение закрывает отправляющий сокет.

При многоадресной рассылке передающее приложение получает *WSA\_QOS\_RECEIVERS* всякий раз, когда число приемников изменяется и отлично от 0. Другими словами, при многоадресной рассылке отправитель получает *WSA\_QOS\_RECEIVERS* всякий раз, когда приемник QoS присоединяется к группе или выходит из группы, до тех пор пока остается хотя бы один приемник.

**Уведомления *WSA\_QOSSENDERS* и *WSA\_QOS\_NOSENDERS*.** Уведомление отправителя аналогично событию приемника, за исключением того, что связано с приемом сообщения PATH. При одноадресной рассылке после начала работы прием первого сообщения PATH генерирует *WSA\_QOS\_SENDERS*, в то время как сообщение PATH инициирует сообщение *WSA\_QOS\_NO\_SENDERS*.

Аналогичным образом при многоадресной рассылке приемники получают уведомление *WSA\_QOS\_SENDERS* всякий раз, когда число отправителей изменяется и отлично от 0. Когда число отправителей доходит до 0, приложению передается сообщение *WSA\_QOS\_NO\_SENDERS*.

**Уведомление *WSAQOSREQUESTCONFIRMED*** — последнее сообщение о состоянии. С его помощью приложения QoS уведомляются о подтверждении запроса на резервирование. В структуре *RSVP\_STATUSINFO* имеется поле *ConfirmRequest*. Если его значение отлично от 0, поставщик службы QoS уведомляет приложение о подтверждении запроса на резервирование. Этот объект — параметр, относящийся к поставщику, он может передаваться вместе со структурой QoS команде *SIO\_SET\_QOS*.

## Шаблоны QoS

В Winsock есть несколько предопределенных структур QoS, называемых шаблонами, которые приложение может вызвать по имени. Эти шаблоны



определяют параметры QoS для нескольких широко используемых звуковых и видеокодов, таких как G711 и H263QCIF. Функция *WSAGetQOSByName* определена так:

```
BOOL WSAGetQOSByName(
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);
```

Если вы не знаете имен имеющихся шаблонов, можете воспользоваться этой функцией для их перебора. Для этого выделите достаточно большой буфер в *lpQOSName*, присвойте его первому символу значение 0 и передайте нулевой указатель в *lpQOS*:

```
WSABUF wbuf;
char cbuf[1024];

cbuf[0] = 0;
wbuf.buf = cbuf;
wbuf.len = 1024;
WSAGetQOSByName(s, &wbuf, NULL);
```

По возвращении буфер символов содержит массив строк, разделенных нулевыми символами. Список завершается еще одним нулевым символом. Это означает, что последняя строка будет иметь два последовательных нулевых символа. Из этого массива можно получить имена всех имеющихся шаблонов и запросить один из них. Следующий код реализует поиск шаблона G711:

```
QOS qos;
WSABUF wbuf;

wbuf.buf = "G711";
wbuf.len = 4;
WSAGetQOSByName(s, &wbuf, &qos);
```

Если запрашиваемого шаблона не существует, поиск возвращает *FALSE* и генерируется ошибка *WSAEINVAL*. В случае успеха функция возвращает *TRUE*. Пример *Qostemplate.c* на прилагаемом компакт-диске показывает, как перебирать имеющиеся шаблоны QoS.

Кроме того, вы можете создать свой собственный шаблон QoS, чтобы другие приложения могли запрашивать его по имени. Здесь задействованы две функции — *WSCInstallQOSTemplate* и *WSCRemoteQOSTemplate*. Первая — создает шаблон QoS, вторая — его удаляет. Вот прототипы этих функций:

```
BOOL WSCInstallQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
```

```

BOOL WSCRemoveQOSTemplate( ^(*, ^ацм^ >1Ци *у>$ftft,-- W
const LPGUID IpProviderId,^ ,лш ^ш', <MSt татада -* i
LPWSABUF IpQOSName * *

```

Описание этих функций очевидно. Для создания шаблона вызовите *WSCInstallQOSTemplate* с GUID именем шаблона и параметрами QoS. GUID — это уникальный идентификатор для этого шаблона, который может быть сгенерирован с помощью таких утилит, как *Uuidgen.exe*.

Для удаления шаблона просто передайте его имя вместе с тем же GUID, что использовался при его задании, функции *WSCRemoveQOSTemplate*. В случае успеха обе функции возвращают *TRUE*.

## Примеры

Рассмотрим примеры программ, использующих QoS. Первый пример с TCP — наиболее простой, потому что требует соединения. Второй — использует UDP без каких-либо вызовов соединений. В последнем примере показано применение многоадресного UDP. Во всех трех примерах мы воспользуемся функцией *WSAEventSelect*, поскольку она проще *WSAAsyncSelect*. Здесь приведен полный пример программы для одноадресной рассылки TCP и только важнейшие фрагменты программ для UDP и многоадресной рассылки. Дело в том, что многие концепции одинаковы, независимо от типа используемого сокета. Полные примеры программ находятся на прилагаемом компакт-диске. Все три примера основаны на вспомогательных процедурах *PrintQos* и *FindProtocolInfo*, описанных в файлах *Printqos.c* и *Provider.c*. Первая — просто печатает содержимое структуры QoS, вторая — находит протокол из каталога поставщика с требуемыми атрибутами, такими как QoS.

## Одноадресный TCP

Пример одноадресного TCP приведен в листинге 12-1. Код для этого примера находится в папке Chapter 12 в файле *Qostcp.c*. Этот пример объемный, но не очень сложный. Основная часть кода — обычный код *WSAEventSelect*. Единственное исключение — это действия в случае события *FD\_QOS*. Главная функция не делает ничего необычного. Аргументы анализируются, создается сокет и вызывается функция *Server* или *Client*, в зависимости от того, как используется приложение: в качестве сервера или клиента. Сначала рассмотрим клиентское соединение.

Во всех трех примерах параметр командной строки сообщает, когда включать QoS: до, во время, после соединения или после того, как партнер запросит локальное включение QoS. Перечислим эти параметры:

- **-q:[b,d,a,e]** — задает QoS до (b), во время (d) или после (a) или ожидает события *FD\_QOS* (e) до задания;
- **-s** — работает как сервер;
- **-c:Server-IP** — работает как клиент и подключается к серверу по заданному адресу;

И -w — для передачи данных ожидает приема сообщения RESV;

III -г — задает параметр для получения уведомления после подтверждения резервирования.

Если QoS включается до соединения (для клиента), привяжите сокет к произвольному порту и затем вызовите *SIOSET\_QOS* с отправляющей структурой *FLOWSPEC*. Заметьте: осуществлять привязку до вызова *SIO\_SET\_QOS* не обязательно, поскольку адрес партнера неизвестен до вызова соединения и RSVP-сеанс пока не может быть иницирован.

Если пользователь включает QoS во время соединения, структура *QOS* передается вызову *WSAConnect*. Этот вызов иницирует сеанс RSVP и подключает клиента к заданному серверу. (Можно избрать и другой способ: чтобы программа ожидала, пока партнер не включит QoS и структура QoS не будет передана *WSAConnect*!) Код берет отправляющую структуру QoS, передает результат логической операции ИЛИ над флагом *SERVICE\_NO\_QOSSIGNALING* полю *ServiceType* в структурах *FLOWSPEC* и вызывает *WSAIoctl* с *ioctl*-командой *SIO\_SET\_FLAG*. Тем самым дается указание поставщику службы QoS не вызывать TC, но продолжать поиск сообщений RSVP.

После включения QoS регистрируются события, которые хочет получать клиент, включая *FD\_QOS*. Заметьте: QoS нужно включить на сокете заранее, чтобы приложение запросило принимающее *FD\_QOS*. После этого клиент ожидает в цикле по *WSAWaitForMultipleEvents*, который завершается, когда одно из выбранных событий свободно. Когда происходит событие, события перечисляются вместе с любыми ошибками в *WSAEnumNetworkEvents*.

В основном *Qostcp.c* обрабатывает другие события, типа *FD\_READ*, *FD\_WRLTE* и *FD\_CLOSE* (См. пример программы с *WSAEventSelect* в главе 8). Но обратите внимание на событие *FD\_WRITE*. Один из параметров командной строки ожидает приема RSVP-сообщения PATH до передачи данных. Это особенно важно, если передаваемые данные могут превысить негарантированную пропускную способность сети. Функция *AbleToSend* вызывает *SIO\_CHK\_QOS*, чтобы определить, находятся ли запрашиваемые параметры QoS в доступных негарантированных пределах. Если это так, можно отправлять данные, если нет — следует ожидать подтверждения.

В рассматриваемом случае клиента мы хотим получить сообщение *WSA\_QOS\_RECEIVERS*, указывающее на прием сообщения RESV при получении события *FD\_QOS*. На этом этапе мы вызываем команду *SIO\_CHK\_QOS*, чтобы получить информацию о состоянии. Флаг *WSA\_QOS\_RECEIVERS* может быть возвращен двумя способами. Во-первых, в поле *iErrorCode* структуры *WSANETWORKEVENTS* как элемент, проиндексированный по полю *FDJOS\_BLT*. Второй вариант — структура *RSVP\_STATUSINFO* возвращается в буфере, переданном функции *WSAIoctl* с помощью *ioctl*-команды *SIO\_GET\_QOS*. Эта структура также может содержать флаг *WSA\_QOS\_RECEIVERS* в поле *Status-Code*. Если было задано ожидание передачи флага, мы проверяем поле ошибки *WSANETWORKEVENTS*, чтобы определить, была ли возвращена структура *RSVPJSTATUS INFO*. Если флаг присутствует, отправляем данные.

Часть программы, касающаяся сервера, сложнее, но только потому, что обрабатывает несколько клиентских соединений или их отсутствие. Прослу-

шивающий сокет и клиентские сокеты обрабатываются в одном массиве с именем *sc*. Нулевой элемент массива — это прослушивающий сокет, остальные — возможные клиентские соединения. Глобальная переменная *nConns* содержит число имеющихся клиентов. По окончании клиентского соединения все активные сокеты сдвигаются к началу массива. Существует также соответствующий массив описателей событий.

Сначала сервер привязывает прослушивающий сокет, и если пользователь решает включить QoS до приема клиентских соединений, включает прием QoS. Любые параметры QoS, заданные на прослушивающем соquete, копируются в клиентское соединение (если сервер не использует *AcceptEx*). Прослушивающий сокет регистрируется для приема только событий *FDACCEPT*.

Оставшаяся часть процедуры сервера — это цикл, ожидающий события в массиве описателей сокетов. Сначала единственный сокет в массиве — прослушивающий, но по мере установки клиентских соединений будет появляться больше сокетов и соответствующих им событий. Если цикл *WSAWaitForMultipleEvents* завершается по причине происшедшего события и описатель события в массиве указывает на нулевой элемент, то событие происходит на прослушивающем соquete. В этом случае программа вызывает *WSAEnumNetworkEvents*, чтобы выяснить, какое событие происходит. Если событие происходит на клиентском соquete, программа вызывает процедуру обработки *HandleClientEvents*.

Обратите внимание на событие *FD\_ACCEPT* на прослушивающем соquete. Когда оно происходит, вызывается *WSAAccept* с условной функцией. Помните: параметры QoS, передаваемые в условную функцию, не надежны, и если в Windows 98 такой параметр отличен от 0, нужно задать какую-нибудь QoS. Windows 2000 не имеет этого ограничения, и QoS можно включить в любой момент. Если пользователь решает включить QoS во время принимающего вызова, это происходит в условной функции. Когда клиентский сокет принят, создается соответствующий описатель события, и события регистрируются для сокета.

Функция *HandleClientEvents* обрабатывает любые события, происходящие на клиентских сокетах. События чтения и записи очевидны; единственный вопрос — нужно ли ожидать подтверждения резервирования до передачи данных. Если пользователь решит, что да, клиент ожидает возвращения сообщения *WSA\_QOS\_RECEF/ERS* в событии *FD\_QOS*. Если сообщение возвращается, передача данных не начинается до приема *FD\_QOS*. Наиболее важная часть этого примера — включение QoS на соquete и обработка *FD\_QOS*.

### Листинг 12-1. Пример одноадресного TCP (*Qostcp.c*)

```
// Модуль: Qostcp.c
//
#include <winsock2.h>
#include <windows.h>
#include <qos.h>
#include <qossp.h>
```

**Листинг 12-1. (продолжение)**

```

«include "provider.h"
«include "printqos.h"

«include <stdio.h>
«include <stdlib.h>

«define QOS_BUFFER_SZ      16000 // Размер буфера по умолчанию для
                                // SIO_GET_QOS
«define DATA_BUFFER_SZ    2048 // Размер буфера для передачи/приема данных

«define SET_QOS_NONE       0    // QOS отсутствует
«define SET_QOS_BEFORE     1    // Включить QOS на прослушивающем сожете
«define SET_QOS_DURING     2    // Включить QOS в условном приеме
«define SET_QOS_AFTER      3    // Включить QOS после приема
«define SET_QOS_EVENT      4    // Дождаться FD_QOS и затем включить

«define HAX_CONN           10

int  iSetQos,              // Когда включить QOS?
     nConns;
BOOL bServer,             // Клиент или сервер?
     bWaitToSend,         // Ожидать передачи данных, пока не выполнится RESV
     bConfirmResv;
char szServerAddr[64];    // Адрес сервера
QOS  clientQos,           // Структура клиента QOS
     serverQos;           // Структура сервера QOS
RSVP_RESERVE_INFO  qosreserve;

// Создать несколько общих структур FLOWSPEC
//
const FLOWSPEC flowspec_notraffle = {QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     SERVICETYPE.NOTRAFFIC,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED};

const FLOWSPEC flowspec_g711 = {8500,
                                680,
                                17000,
                                QOS_NOT_SPECIFIED,
                                QOS_NOT_SPECIFIED,
                                SERVICETYPE.CONTROLLEDLOAD,
                                340,
                                340};

```

Листинг 12-1. (продолжение)

```
const RCVSPEC flowspec_guaranteed {17000, iQn xvteaefl.sor-tevist
1260, hfve
34000, ( EV i ' 198
QOS_NOT_SPECIFIED, Еам ' tt
QOS_NOT_SPECIFIED, ,16.-.
SERVICETYPE.GUARANTEED,
340,
340};

* '8* ) stpeeu • \\

// функция: SetReserveInfo W
// Описание: 1NA0 \\
// Для приемников: если требуется подтверждение, это должно быть сделано W
// с помощью структуры RSVP_RESERVE_INFO su blov

void SetQosReserveInfo(QOS «lpqos)

gosreserve.ObjectHdr.ObjectType = RSVP_OBJECT_RESERVE_INFO;
gosreserve.ObjectHdr.ObjectLength = sizeof(RSVp_RESERVE_INFO);
gosreserve.Style = RSVP_DEFAULT_STYLE;
gosreserve.ConfirmRequest = bConfirmResv; ...
gosreserve.NumPolicyElement = 0; 'ievia, o- III
gosreserve.PolicyElementList = NULL; (> ' ж-
gosreserve.FlowDescList = NULL; " ' i- nq
J:x3

lpqos->ProviderSpecific.buf = (char *)&qosreserve;
lpqos->ProviderSpecific.len = sizeof(qosreserve);

return; \\
:e<H>ot*n0
\\
\\
\\
// функция: InitQos \\
// Описание: \\
// Создает структуры клиента и сервера QOS. Это выделено в отдельную функцию,
// чтобы вы могли изменять запрашиваемые параметры QOS
// и смотреть, как это повлияет на приложение. \\

void InitQos0
{
clientQos.SendingFlowspec = flowspec_g711;
clientQos.ReceivingFlowspec = flowspec_nottraffic;
clientQos.ProviderSpecific.buf = NULL;
clientQos.ProviderSpecific.len = 0;

serverQos.SendingFlowspec = flowspec_nottraffic;
-3d
```

## Листинг 12-1. (продолжение)

IMNT»NR

```

serverQos.ReceivingFlowspec = flowspec_g711;
serverQos.ProviderSpecific.buf = NULL;
serverQos.ProviderSpecific.len = 0;
if (bConfirmResv)
    SetQosReserveInfo(&serverQos);

```

```

// Функция: usage                                i*e-
//
// Описание:
// Печать информации об использовании
void usage(char progame)
{
    printf("usage  XS-q:x-s -c:IP\n", progame);
    printf("  -q:[b,d,a,e] When to request QOS\n");
    printf("    b      Set QOS before bind or connect\n");
    printf("    d      Set QOS during accept condfunc\n");
    printf("    a      Set QOS after sessionsetup\n");
    printf("    e      Set QOS only upon receipt of FD_QOS\n");
    printf("  -s      Act asserver\n");
    printf("  -c:Server-IP Act asclient\n");
    printf("  -w      Wait to send until RESV hasarrived\n");
    printf("  -r      Confirm reservation request\n");
    ExitProcess(-i);
}

```

```

// Функция: ValidateArgs                          n-i.li
// Описание:
// Анализирует аргументы командной строки и задает глобальные переменны*,
// чтобы определить, как должно работать приложение
void ValidateArgs(int argc, char **argv)
{
    int i;
    // Инициализация глобальных переменных значениями по умолчанию ^"wo n
    iSetQos = SET_QOS_NONE;
    bServer = TRUE;
    bWaitToSend = FALSE;
    bConfirmResv = FALSE;
    for(i = 1; i < argc;
    {
        if ((argv[i][0] == '-') (argv[i][0]

```

Листинг 12-1. (продолжение)

K Rtt(vOK\rO

```

switch (tolower(argv[i][1]))
{
    case 'q': // Когда включать QOS
        if (tolower(argv[i][3]) == 'b')
            iSetQos = SET_QOS_BEFORE;
        else if (tolower(argv[i][3]) == 'd')
            iSetQos = SET_QOS_DURING;
        else if (tolower(argv[i][3]) == 'a')
            iSetQos = SET_QOS_AFTER;
        else if (tolower(argv[i][3]) == 'a')
            iSetQos = SET_QOS_EVENT;
        else
            usage(argv[0]);
        break;

    case 's': // Сервер
        printf("Server flag set!\n");
        bServer = TRUE;
        break;

    case 'c': // Клиент
        printf("Client flag set!\n");
        bServer = FALSE;
        if (strlen(argv[i]) > 3)
            strcpy(szServerAddr, &argv[i][3]);
        else
            usage(argv[0]);
        break;

    case 'w': // Не отправят до?
        // пока не поступило сообщение
        bWaitToSend = TRUE;
        break;

    case 'r':
        bConfirmResv = TRUE;
        break;

    default:
        usage(argv[0]);
        break;
}

return;

```

// Функция: AbleToSend

f)

// Описание:

// Проверяет, могут ли данные быть отправлены на сокет до поступления

си. след. стр.



## Листинг 12-1. (продолжение)

И сообщений RESV. Эта функция проверяет, достаточен ли имеющийся в настоящий момент в сети негарантированный уровень для уровней QOS, заданных на сожете.

```
//
BOOL AbleToSend(SOCKET s)
{
    int ret;
    DWORD dwCode = ALLOWED_TO_SEND_DATA,
           dwValue,
           dwBytes;

    ret = WSAIoctl(s, SIO_CHK_QOS, idwCode, sizeof(dwCode),
                  &dwValue, sizeof(dwValue), &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return FALSE;
    }
    return (BOOL)dwValue;
}
```

// Функция: ChkForQosStatus

//  
 // Описание'  
 // Проверяет наличие объекта RSVP\_STATUS\_INFO и определяет,  
 // имеются ли в нем нужные флаги

```
//
DWORD ChkForQosStatus(QOS *lpqos, DWORD dwFlags)
{
    QOS_OBJECT_HDR *objhdr = NULL;
    RSVP_STATUS_INFO «status = NULL;
    char " " .bufptr = NULL;
    BOOL bDone = FALSE;
    DWORD obicount = 0,

    if (lpqos->ProviderSpecific.len == 0)
        return 0;

    bufptr = lpqos->ProviderSpecific.buf;
    objhdr = (QOS_OBJECT_HDR *)bufptr;

    while (ibDone)
    {
        if (objhdr->ObjectType == RSVP_OBJECT_STATUS_INFO)
        {
            status = (RSVP_STATUS_INFO *)objhdr;
            if (status->StatusCode & dwFlags)
                return 1;
        }
    }
}
```

## Листинг 12-1. (продолжение)

```

    }
    else if (objhdr->ObjectType == QOS_OBJECT_END_OF_LIST)
        bDone = TRUE;

    bufptr += objhdr->ObjectLength;
    objcount += objhdr->ObjectLength;
    objhdr = (QOS_OBJECT_HDR *)bufptr;

    if (objcount >= lpqos->ProviderSpecflc.l>n)
        bDone = TRUE;

    return 0;

// Функция. HandleClientEvents
//
// Описание'
// Эта функция вызывается функцией Server для обработки событий,
// которые произошли на описателях клиентов SOCKET. Массив
// сокетов передается вместе с массивом событий и индексом клиента,
// получившего оповещение. В функции событие декодируется,
// и выполняется соответствующее действие.

void HandleClientEvents(SOCKET socks[], HANDLE events[], int index) {
    WSANETWORKEVENTS ne;
    char databuf[4096];
    WSABUF wbuf;
    DWORD dwBytesRecv, dwFlags;
    int ret, i;

    // Перебор происшедших в сети событий
    //
    ret = WSAEnumNetworkEvents(socks[index], events[index], 4ne);
    if (ret == SOCKET.ERROR)

        pnntf("WSAEnumNetworkEvents() failed: Xd\n",
            WSAGetLastError0);
        return;

    // Данные для чтения
    if ((ne INetworkEvents & FD.READ) == FD_READ)
        wbuf.buf = databuf,

```

см. след. стр.

**Листинг 12-1.**    *{продолжение}*      ..«ЧТИХ      Л      июни.

```

1-      -t t m N <      III!      <<re
if (ne.iErrorCode[FD_READ_BIT])      isoci
    printf("FO_READ error: Xd\n",
        ne.iErrorCode[FD_READ_BIT]);      -bAtvo •• TJctud
else      . (do p+
    printf("FD_READ\n");      O_bB<)

dwFlags = 0;      •~<; tnuootdo) U
ret = WSAREcv(socks[index], &wbuf, 1, &dwBytesRecv, ifft * *no<Jd
    &dwFlags, NULL, NULL);      {
if (ret == SOCKET_ERROR)      j, j;      ;0 muJei

    printf("WSAREcv() failed: Xd\n", WSAGetLastError());
    return;      *      ',

wbuf.len = dwBytesRecv;      \\

printf("Read: %d bytes\n", dwBytesRecv);      >      ,ф втв      V

// Может записывать данные; здесь никакие действия не предпринимаютздхоо      л

if ((ne.lNetworkEvents & FD_WRITE) == FD_WRITE)
{
    if (ne.iErrorCode[FD_WRITE_BIT])      F4,/iev3tneJ:I3eIbnBH bxov
        printf("FD_WRITE error: Xd\n",
            ne.iErrorCode[FD_WRITE_BIT]);      ;m
    else      ,-f»e0*]tudeJrb
        printf("FD_WRITE\n");

// Клиент закрыл соединение. Закрываем сокет с
// и очищаем структуры данных.      jm      jni
//      ц
if ((ne.lNetworkEvents & FD_CLOSE) == FD.CLOSE)
{
    if (ne.iErrorCode[FD_CLOSE_BIT])      i wt      • o s x«йbP»a«очл ччввчвП \\
        printf("FD_CLOSE error: J!d\n",w|j8H3Oi)eJneV'      \\
            ne.iErrorCode[FO_CLOSE_BIT]);      {R0Я
    else      >
        pnntf("FD_CLOSE ... \n");      .6» ( )s?n«v3)!iowj'      'A8Y("      • Utq
        closesocket(socks[index]);      iCO^o-      Jit
        WSACloseEvent(events[index]);      ijei

        socks[index] = INVALID_SOCKET;
        //
        // Удаляем элемент клиентского сокета из массива и
        // сдвигаем оставшиеся элементы к началу массива.
        //
        for(i = index; i < MAX_CONN - 1;

```

## Листинг 12-1. (продолжение)

```

        socks[i] = socks[i + 1];
        nConns--;
    }

^ II Получено событие FD_QOS. Это может иметь разные значения.

if ((ne.lNetworkEvents & FD_QOS) == FD.QOS)

    char    buf[QOS_BUFFER_SZ];
    QOS      *lpqos = NULL;
    DWORD     dwBytes;

    .....

3i else
G*    printf("FD_QOS\n");
    »
    lpqos = (QOS) Obuf;
*** lpqos->ProviderSpecific.buf = &buf[sizeof(QOS)];
    lpqos->ProviderSpecific.len = sizeof(buf) - sizeof(COW);

    ret = WSAIoctl(socks[index], SIO.GET.QOS, NULL, 0,
        buf, QOS.BUFFER.SZ, &dwBytes, NULL, NULL);
    if (ret == SOCKET.ERROR)

        printf("WSAIoctl(SIO_GET_QOS) failed M^J4,
            WSAGetLastError());
        return;
    }
PrintQos(lpqos);
//
// Проверим, имеется ли настройка для приема только событий FD.QOS.я1
// Если это так, то нужно вызвать QOS на соединении сейчас; иначе
// клиент никогда не получит сообщение RESV.
if (iSetQos == SET_QOS_EVENT)

    lpqos->ReoeivingFlowspec.ServiceType =
        serverQos.ReceivingFlowspec.ServiceType;

    ret = WSAIoctl(socks[index], SIO.SET.QOS, lpqos,
        dwBytes, NULL, 0, AdwBytes, NULL, NULL);
    if (ret == SOCKET.ERROR)

        printf("WSAIoctl(SIO_SET_QOS) failed: Xd\n",
            WSAGetLastError());
        return;

```

Листинг 12-1.    {продолжение}

```

//
// Изменить lSetQos, чтобы не пришлось снова задавать QOS
// при получении еще одного события FD_QOS
//
lSetQos = SET_QOS_BEFORE;
}
return;

// Функция: SrvCondAccept
// Описание.
// Это основная функция для WSAAccept. Поставщик службы QOS
// имеет ограничение: значения QOS, передаваемые сюда, не надежны,
// поэтому параметр SET_QOS_DURING бесполезен, если мы не вызовем
// SIO_SET_QOS с нашими значениями (в противоположность значениям,
// запрашиваемым клиентом, поскольку те значения должны быть
// возвращены в IpSQOS).
// Заметьте' если в Windows 98 значение IpSQOS отлично от NULL,
// вы должны задать некоторые значения QOS (с помощью SIO_SET_QOS)
// в условной функции, иначе функция WSAAccept не отработает.

mt CALLBACK SrvCondAccept(LPWSABUF lpCallerId,
LPWSABUF lpCallerData, LPQOS IpSQOS, LPQOS lpGQOS,
LPWSABUF lpCalleeId, LPWSABUF lpCalleeData, GROUP *g,
DWORD dwCallbackData)
{
    DWORD dwBytes = 0;
    SOCKET s = (SOCKET)dwCallbackData;
    SOCKADDR_IN client;
    int ret;

    if (nConns == MAXCONN)
        return CF_REJECT;

    memcpy(&client, lpCallerId->buf, lpCallerId->len);
    printf("Client request: Xs\n", inet_ntoa(client.sin_addr));

    if (lSetQos == SET_QOS_EVENT)
    {
        printf("Setting for event!\n");
        serverQos.SendingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING,
        serverQos.ReceivingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;

        ret = WSAIoctl(s, SIO_SET_QOS, &serverQos,
            sizeof(serverQos), NULL, 0, &dwBytes, NULL, NULL);
    }
}

```

## Листинг 12-1. {продолжение}

```

    if (ret == SOCKET_ERROR)
    {
        //J
        printf("WSAIoct1() failed: %d\n",
            WSAGetLastError());
        return CF.REJECT;

return CF.ACCEPT;

// Функция: Server
//
// Описание'
// Эта подпрограмма сервера обрабатывает входящие клиентские соединения.
// Сначала она задает прослушивающий сокет, затем задает QOS
// в нужный момент времени и ожидает входящих клиентов и события. }-ц
//
void Server(SOCKET s)

    SOCKET          sc[MAX_CONN + 1];
    WSAEVENT         hAllEvents[MAX_CONN+1];
    SOCKADDR_IN      local,
                    client;
    mt               clientsz,
                    ret,
                    i;
    DWORD            dwBytesRet;
    WSANETWORKEVENTS ne;

    // Инициализация массивов неверными значениями.
    //
    for(i = 0; i < MAX_CONN+1;

        hAllEvents[i] = WSA_INVALID_EVENT;
        sc[i] = INVALID_SOCKET;

// Нулевой элемент массива будет прослушивающим сокетом.
//
hAllEvents[0] = WSACreateEvent();
sc[0] = s;
nConns = 0,

local.sin_family = AF_INET;
local sin_port = htons(5150);
local sm_addr.s_addr = htonl(INADDR_ANY);

if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)

```

см след стр

## Листинг 12-1. (продолжение)

```

{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;

listen(s, 7);

if (iSetQos == SET_QOS_BEFORE)
{
    ret = WSAIoctl(sc[0], SIO_SET_QOS, &serverQos,
        sizeof(serverQos), NULL, 0, &dwBytesRet, NULL, NULL);
    if (ret == SOCKET_ERROR)

        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;

}
printf("Set QOS on listening socket:\n");
PrintQos(&serverQos);

if (WSAEventSelect(sc[0], hAHEvents[0], FD_ACCEPT) ==
    SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;

while (1)
{
    ret = WSAWaitForMultipleEvents(nConns+1, hAHEvents, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleObject() failed: %d\n",
            WSAGetLastError());
        return;
    }
    if ((i = ret - WSA_WAIT_EVENT_0) > 0) // Сетевое событие клиента I \
        HandleClientEvents(sc, hAHEvents, i); \
    else
    {
        ret = WSAEnumNetworkEvents(sc[0], hAllEvents[0],
            &ne);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAEnumNetworkEvents() failed: %d\n",
                WSAGetLastError());
            return;

```

**Листинг 12-1.** (продолжение')

```

        if ((ne.lNetworkEvents & FD_ACCEPT) == FD_ACCEPT) {
            if (ne.iErrorCode[FD_ACCEPT_BIT]) {
                printf("FD_ACCEPT error: Xd\n",
                    ne.iErrorCode[FD_ACCEPT_BIT]);
            } else {
                printf("FD_ACCEPT\n");
            }

            clientsz = sizeof(client);
            sc[++nConns] = WSAAccept(s, (SOCKADDR *)&client, &clientsz, SrvCondAccept, sc[nConns]);
            if (sc[nConns] == SOCKET_ERROR) {
                printf("WSAAccept() failed: Xd\n", WSAGetLastError());
                nConns--;
                return;

                haHEvents[nConns] = WSACreateEvent();

                Sleep(10000);
                if (iSetQos == SET_QOS_AFTER) {
                    ret = WSAIoctl(sc[nConns], SIO_SET_QOS, &serverQos, sizeof(serverQos), NULL, 0, &dwBytesRet, NULL, NULL);
                    if (ret == SOCKET_ERROR) {
                        printf("WSAIoctl() failed: Xd\n", WSAGetLastError());
                        return;

                        ret = WSAEventSelect(sc[nConns], haHEvents[nConns], FD_READ | FD_WRITE | FD_CLOSE | FD_QOS);
                        if (ret == SOCKET_ERROR) {
                            printf("WSAEventSelect() failed: Xd\n", WSAGetLastError());
                            return;

                            if (ne.lNetworkEvents & FD_CLOSE) {
                                printf("FD_CLOSE\n");
                            }
                            if (ne.lNetworkEvents & FD_READ) {
                                printf("FD_READ\n");
                            }
                            if (ne.lNetworkEvents & FD_WRITE) {
                                printf("FD_WRITE\n");
                            }
                        }
                    }
                }
            }
        }
    }
}

```



## Листинг 12-1. (продолжение)

```

printf("FD_WRITE");  ШОА.C
if (ne.NetworkEvents & FtyMD '
    printf("FD_QOS\n");  18_74300
    8/b2 .10718  1")7Jn112
    18_74300A_07]b000 1131 on
return;

// Функция: Client
// Описание:
// Подпрограмма клиента иницирует соединение, задает Qof в нужны! момент
// времени и обрабатывает входящие события.

void Client(SOCKET s)
{
    SOCKADDR_IN  server,
                local;      ;()tfievls
    WSABUF       wbuf;
    DWORD        dwBytes,
                dwBytesSent,
                dwBytesRecv,
                dwFlags;
    HANDLE        hEvent;
    int           ret, i;
    char          databuf[DATA_BUFFER_SZ];  T3Ж»
    QOS           *lpqos;
    WSANETWORKEVENTS ne;

    hEvent = WSACreateEvent();
    if (hEvent == NULL)

        printf("WSACreateEvent() failed: Xd\n", WSAGetLastErrorfJ);
        return;

    lpqos = NULL;
    if (iSetQos == SET_QOS_BEFORE)

        local.sin_family = AF_INET;
        local.sin_port = htons(0);
        local.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (SOCKADDR *)&local, sizeof(local)) ==
        SOCKET_ERROR)

        printf("bind() failed: Xd\n", WSAGetLastError0);
        return;

```

Листинг 12-1. (продолжение)

```

ret = WSALocatl(s, SIO_SET_QOS, iclientQos,
    sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
if (ret == SOCKET_ERROR)
    printf("WSALocatl(SIO_SET_QOS) failed: %d\n",
        WSAGetLastErrorO);
return;

>
else if (iSetQos == SET_QOS_DURING)
    lpqos = &clientQos;
else if (iSetQos == SET_QOS_EVENT)
{
    clientQos.SendingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;
    clientQos.ReceivingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;

    ret = WSALocatl(s, SIO_SET_QOS, &clientQos,
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
        printf("WSALocatl() failed: %d\n", WSAGetLastErrorO);
    return;

server.sin_family = AF_INET;
server.sin_port = htons(5150);
server.sin_addr.s_addr = inet_addr(szServerAddr);

printf("Connecting to: %s\n", lnet_ntoa(server.sin_addr));

ret = WSAConnect(s, (SOCKADDR *)server, sizeof(server),
    NULL, NULL, lpqos, NULL);
if (ret == SOCKET_ERROR)
    printf("WSAConnect() failed: %d\n", WSAGetLastErrorO);
    return;

ret = WSAEventSelect(s, hEvent, FD_READ | FD_WRITE
    | FD_CLOSE | FD_QOS);
if (ret == SOCKET_ERROR)
    printf("WSAEventSelect() failed: %d\n", WSAGetLastErrorO);

```

см. след. стр.

**Листинг 12-1.**    *(продолжение)*

```

\.\>
return;

}

wbuf.buf = databuf;
wbuf.len = DATA_BUFFER_SZ;

memset(databuf, '#', DATA_BUFFER_SZ);
databuf[DATA_BUFFER_SZ-1] = 0;

while (1)
{
    ret = WSAWaitForMultipleEvents(1, hEvent, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WAIT_FAILED)
        printf("WSAWaitForMultipleEvents failed: %d\n",
            GetLastError());
    return;

    ret = WSAEnumNetworkEvents(s, hEvent, &ne);
    if (ret == SOCKET_ERROR)
        printf("WSAEnumNetworkEvents() failed: %d\n",
            GetLastError());
    return;

    if (ne.lNetworkEvents & FD_READ)
    {
        if (ne.iErrorCode[FD_READ_BIT])
            printf("FD_READ error: %d\n",
                ne.iErrorCode[FD_READ_BIT]);
        else
            printf("FD_READ\n");

        wbuf.len = 4096;
        dwFlags = 0;
        ret = WSARecv(s, &wbuf, 1, &dwBytesRecv, &dwFlags,
            NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n",
                GetLastError());
            return;
        }
        printf("Read: %d bytes\n", dwBytesRecv);

        wbuf.len = dwBytesRecv;
        ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL,

```

## ГДАВА12

Листинг 12-1. {продолжение'}

шЫхтук) . М1 1и\*т>зд>

```

        NULL);
    if (ret == SOCKET_ERROR)
        printf("WSASend() failed:
            WSAGetLastErrorO);
        return;

    printf("Sent: %d bytes\n", dwBytesSent);

    if (ne.lNetworkEvents & FD.WRITE)
    {
        if (ne.iErrorCode[FD_WRITE_BIT])
            printf("FD_WRITE error: Xd\n",
                ne.iErrorCode[FD_WRITE_BIT]);
        else
            printf("FD_WRITE\n");

        if (!bWaitToSend)

        wbuf.buf = databuf;
        wbuf.len = DATA_BUFFER_SZ;

        // Если сеть не может обеспечить пропускную способность,
        // не отправлять данные.
        //
        if (!AbleToSend(s))

            printf("Network is unable to provide "
                "sufficient best-effort bandwidth\n");
            printf("before the reservation "
                "request is approved\n");

        for(i = 0; i < 1; i++)

            ret = WSASend(s, &wbuf, 1, idwBytesSent, 0,
                NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: Xd\n",
                    WSAGetLastErrorO);
                return;

                printf("Sent: Xd bytes\n", dwBytesSent);
            }
    }

    if (ne.lNetworkEvents & FD.CLOSE)

```

см.след.стр.

Листинг 12-1.    {продолжение}

```

{
    if (ne.iErrorCode[FD_CLOSE_BIT] > 0) {
        printf("FD_CLOSE error: %d\n", ne.iErrorCode[FD_CLOSE_BIT]);
    }
    else {
        printf("FD_CLOSE ... \n");
        closesocket(s);
        WSACloseEvent(hEvent);
        return;
    }

    if (ne.lNetworkEvents & FD_QOS) {

        char        buf[QOS_BUFFER_SZ];
        QOS         *lpqos = NULL;
        DWORD        dwBytes;
        BOOL         bRecvRESV = FALSE;

        if (ne.iErrorCode[FD_QOS_BIT]) {
            printf("FD_QOS error: %d\n", ne.iErrorCode[FD_QOS_BIT]);
            if (ne.iErrorCode[FD_QOS_BIT] == WSA_QOS_RECEIVER5)
                bRecvRESV = TRUE;
        }
        else {
            printf("FD_QOS\n");

            lpqos = (QOS *)buf;
            ret = WSAIoctKs, SIO_GET_QOS, NULL, 0,
                buf, QOS_BUFFER_SZ, idwBytes, NULL, NULL);
            if (ret == SOCKET_ERROR) {
                printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
                    WSAGetLastError());
                return;
            }

            PrintQos(lpqos);

            // Проверить, возвращен ли объект состояния в структуре QOS,
            // которая может также содержать флаг WSA_QOS_RECEIVERS
            //
            if (ChkForQosStatusdpqos, WSA_QOS_RECEIVERS))
                bRecvRESV = TRUE;

            if (iSetQos == SET_QOS_EVENT) {
                lpqos->SendingFlowspec.ServiceType =
                    clientQos.SendingFlowspec.ServiceType;
                ret = WSAIoctKs, SIO_SET_QOS, lpqos, dwBytes,

```

## Листинг 12-1. {продолжение}

```

        NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)

        printf("WSAioctl(SIO_SET_QOS) failed: %d\n",
               WSAGetLastError());
    return;

    // Изменить lSetQos, чтобы не пришлось снова включать QoS
    // при приеме еще одного события FD_QOS.
    //
    iSetQos = SET_QOS_BEFORE;

    if (bWaitToSend && bRecvRESV)
    {
        wbuf.buf = databuf;
        wbuf.len = DATA_BUFFER_SZ;

        for(i = 0; i < 1;

            ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                          NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n",
                       WSAGetLastError());
                return;
            }
            printf("Sent: %d bytes\n", dwBytesSent);

        >
        return;

// Функция: main
//
// Описание:
//     Инициализирует Winsock, анализирует аргументы командой строки, создает
//     сокет QOS TCP и вызывает соответствующую подпрограмму обработки
//     в зависимости от переданных аргументов
//
int main(int argc, char **argv)
{
    WSADATA          wsd;

```

см. след. стр.

**Листинг 12-1.** (продолжение)

```

WSAPROTOCOL_INFO *pinfo = NULL;
SOCKET s;

// Анализ командной строки
ValidateArgs(argc, argv);
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    printf("Unable to load Winsock: Xd\n", GetLastError());
    return -1;
}

pinfo = FindProtocolInfo(AF_INET, SOCK_STREAM, IPPROTO_TCP,
    XP1_QOS_SUPPORTED);
if (!pinfo)
{
    printf("unable to find suitable provider!\n");
    return -1;
}

printf("Provider returned: Xs\n", pinfo->szProtocol);

s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
    FROM_PROTOCOL_INFO, pinfo, 0, WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    printf("WSASocket() failed: Xd\n", WSAGetLastError());
    return -1;
}

InitQoS();

if (bServer)
    Server(s);
else
    Client(s);

closesocket(s);
WSACleanup();
return 0;

```

**Одноадресный UDP**^\*  
v

Одноадресный UDP предоставляет больше возможностей по сравнению с TCP. Пример UDP на прилагаемом компакт-диске называется *Qosudp.c*. Он объединяет отправитель и приемник. Отправитель может двумя способами сообщить поставщику службы QoS о том, куда следует отправлять данные. Помните: для инициирования сеанса RSVP требуется адрес партнера. Его можно определить с помощью вызова *WSAConnect* или *SIO\_SET\_QOS* с объектом *QOSJDESTADDR*.

Пример одноадресного UDP также использует параметр, сообщающий, когда необходимо включить QoS. Если пользователь хочет задать QoS до привязки или подключения, используется `ioctl`-команда `SIO_SET_QOS` с объектом `QOSDESTADDR`. Если принято решение включить QoS во время выполнения условной функции `WSAAccept`, параметры QoS указываются в вызове `WSAConnect`. Если пользователь желает включить QoS после установления сеанса, `WSAConnect` вызывается без QoS, а `SIO_SET_QOS` — позднее без объекта `QOSJDESTADDR`. Наконец, чтобы задать QoS только после получения уведомления о событии `FD_QOS`, `SIO_SET_QOS` вызывается с объектом `QOSJDESTADDR`, но над флагом `SERVICE_QOS_NOJSIGNALING` и значением поля `ServiceType` структуры `FLOWSPEC` нужно выполнить логическую операцию ИЛИ.

У принимающей стороны немного параметров. Различные флаги, указывающие, когда задавать QoS, здесь не применяются. QoS задают до приема данных или приемник ожидает, когда произойдет событие `FD_QOS`. Такое поведение вызвано тем, что UDP не получает никакого запроса на подключение: QoS не задана во время выполнения функции приема или после установления сеанса. Приемник имеет также возможность определить другой стиль фильтра, например, фиксированный или общий явный. Если задан другой фильтр, с параметром `-g:IP` должен быть передан IP-адрес. Функция `SetQosReceivers` заполняет структуру `RSVPJIESERVEINFO` структурой `RSVP_FILTERSPEC`, определяющей IP-адрес отправителя. Задать номер порта отправителя особенно важно. Это означает, что приемник должен знать IP-адрес каждого отправителя и номер порта, к которому он привязан.

Приемник может также использовать `WSAConnect`, чтобы сопоставить IP-адрес отправителя сокету. Но поскольку приемник UDP вправе задать разные стили фильтров и количество отправителей, `WSAConnect` использовать нельзя. Помните: если `WSAConnect` применяется для сопоставления IP-адреса конечной точки, операции передачи и приема правомерны только для этого партнера, и ему должна быть сопоставлена QoS.

Пример с одноадресным UDP похож на пример с TCP. Различны лишь способы включения QoS на сокет. Приложения UDP требуют от отправителя задания IP-адреса приемника для вызова `RSVP`, а приложения TCP делают это по умолчанию, в вызове соединения. Цикл событий для обоих приложений почти одинаков. Не забудьте: для приложений UDP сокет должен быть привязан локально до включения любой QoS (передачи или приема) с `SIO_SET_QOS`, когда `WSAConnect` не используется. Допускается привязка к `INADDRANY` порту 0, а также использование специального IP-адреса и порта. `WSAConnect` выполняет неявную привязку, поэтому если QoS включается на этом этапе, делать явную привязку заранее не нужно.

## Многоадресный UDP

Последний пример — это многоадресная QoS (файл `Qosmcast.c` на компакт-диске). Его главная функция — `WSAJoinLeaf`, которую приложение должно вызвать, чтобы присоединиться к группе многоадресной рассылки. После этого оно может также передавать параметры QoS. В примере с многоадресной рассылкой использованы те же параметры, что и в примере с одноадрес-



ресной рассылкой TCP Вы можете выбрать момент задания QoS на сожете. Если решите сделать это во время выполнения условной функции приема, QoS будет передана в вызов *WSAJoinLeaf*. Или задайте QoS с помощью вызова *WSAIoctl* с *SIO\_SET\_QOS*.

Один из параметров для приемника позволяет пользователям задать фиксированный или общий явный фильтр. Помните многоадресный UDP по умолчанию использует фильтр, содержащий метасимволы. Другой тип фильтра применим только к приемникам, и если требуется именно это, адрес каждого отправителя задается в параметре командной строки *-g* *SenderIP*. Пользователи могут задавать фильтры через параметр *-f* с режимом *se* — для общего явного, и с режимом *ff* — для фиксированного фильтра.

Параметр *-t* отправитель и получатель используют, чтобы задать многоадресную группу, к которой следует присоединиться. Этот параметр можно задавать несколько раз и присоединяться к любому количеству групп. Параметр *-s* указывает, что программа функционирует как отправитель. Параметр *-w* сообщает отправителю о необходимости подождать уведомления *WSA\_QOS\_RECEIVERS* по передаче данных. Наконец, параметр *-q* определяет, когда включать QoS, причем независимо от выбранного момента, сокет локально привязывается к порту 5150.

На самом деле, можно выбрать любой порт или задать 0, чтобы порт был выбран автоматически. Между тем, если получатель задает фиксированный или общий явный фильтр, он должен также сообщить IP-адрес и порт отправителя. Для простоты мы используем фиксированный порт. В отличие от одноадресного UDP получатель не должен локально привязывать порт, чтобы задать QoS — ведь *WSAJoinLeaf* неявно привязывает сокет, если он еще не привязан. А вот использовать вместо *WSAJoinLeaf* команды сокета *IP\_ADD\_MEMBERSHIP* и *IP\_DROP\_MEMBERSHIP* нельзя — тогда параметры QoS не будут применяться к сокету.

В этом примере QoS включается почти так же, как и в случае с одноадресным UDP, поэтому мы не будем подробно на этом останавливаться. Главное — представлять, как иницируется сеанс RSVP и что нужно для генерирования сообщений PATH и RSVP.

## ATM и QoS

Как уже упоминалось, служба QoS изначально доступна в сетях ATM. Windows 2000 и Windows 98 (с SP1) поддерживают программирование ATM из Winsock. QoS изначально доступна в сети ATM, так что собственные сеть, приложение и компоненты политики, необходимые для QoS через IP, не требуются. Это же относится и к службе управления доступом (Admission Control Service, ACL) и к протоколу RSVP. Вместо этого, коммутатор ATM выделяет пропускную способность и предотвращает ее избыточное использование.

Помимо этих различий функции Winsock API работают с QoS в сетях ATM немного иначе, чем с QoS поверх IP. Первое главное отличие — запрос пропускной способности QoS обрабатывается, как часть запроса соединения. Это отличается от QoS поверх IP — сеанс RSVP устанавливается отдельно от соединения. Если запрос пропускной способности отклонен в ATM, соеди-

нение установлено не будет. Отсюда следует, что оба базовых поставщика ATM требуют соединения. В результате не возникает проблем с заданием уровней QoS для сокета, не требующего соединения, и последующим определением конечной точки соединения.

Второе и главное отличие — только одна сторона задает параметры QoS для соединения. Это значит, что если клиент хочет задать QoS на соединении, отправляющая и принимающая структуры *FLOWSPEC* будут назначены внутри структуры QoS, переданной функции *WSAConnect*. Эти значения затем применяются к соединению (в QoS через IP отправитель запрашивает определенные уровни QoS и получатель осуществляет необходимое резервирование). Кроме того, прослушивающий сокет может иметь QoS, заданную с помощью *WSAIoctl* и *SIO\_SET\_QOS*, и ее параметры будут применены к любому входящему соединению. Иными словами, QoS должна быть включена во время настройки соединения, нельзя включить QoS на уже установленном соединении.

Отсюда следует, что если QoS включено для соединения, нельзя повторно согласовать параметры QoS с помощью вызова *WSAIoctl* и *SIO\_SET\_QOS*. Заданные параметры QoS останутся неизменными до окончания соединения.

Помните, RSVP отсутствует, и не происходит никакого оповещения. Это означает, что никакие флаги состояния не генерируются, до окончания соединения не происходит никаких уведомлений или событий.

и (

## Резюме

; £,

QoS предлагает мощные возможности приложениям, требующим гарантированного уровня сетевого обслуживания. Установка соединения QoS достаточно сложна, но это не должно вас останавливать. Самое важное — знать, как и когда генерируются сообщения RSVP.

i

J

! I

## Простые сокеты

Простой сокет обеспечивает доступ к базовому протоколу передачи. В этой главе мы рассмотрим использование простых сокетов для моделирования таких утилит IP, как Traceroute и Ping. Простые сокеты также можно применять для манипулирования информацией в заголовке IP. Мы обсудим только протокол IP, поскольку большинство остальных протоколов (кроме ATM) вообще не поддерживает простые сокеты. Все простые сокеты создаются с использованием типа сокета *SOCKRAW* и поддерживаются сейчас только Winsock 2, так что ни Windows CE, ни Windows 95 (без обновления Winsock 2) не могут их использовать.

Работа с простыми сокетами требует достаточно глубокого знания структур протоколов. Подробно в книге мы их рассматривать не будем, а обсудим лишь Internet Control Message Protocol (ICMP), Internet Group Management Protocol (IGMP) и User Datagram Protocol (UDP). С помощью ICMP утилита Ping определяет, эффективен ли маршрут к узлу и отвечает ли тот на запросы. Разработчикам часто требуется программно определить рабочее состояние и доступность компьютера.

IGMP используется многоадресным вещанием IP для оповещения маршрутизаторов о присоединении узлов к группе; недавно в большинство платформ Win32 была добавлена функция поддержки IGMP версии 2. Но вы можете и отправить собственные IGMP-пакеты, чтобы прекратить членство в группе. Мы рассмотрим протокол UDP вкупе с параметром сокета *IPHDRINCL* в качестве способа, позволяющего сделать это.

Для всех трех перечисленных протоколов мы обсудим только аспекты, необходимые для полного объяснения кода. Для более подробного изучения обратитесь к книге Ричарда Стивена'.

### Создание простого сокета

Первый шаг в использовании простого сокета — его создание. Вы можете сделать это, как с помощью функции *socket*, так и *WSASOCKET*. Как правило, записи каталога в Winsock для IP, которым соответствует тип сокета *SOCK RAW*, отсутствуют, но это вам не мешает, так как означает только невозможность создать простой сокет с помощью структуры *WSAPROTOCOLINFO*. (Информация о перечислении записей протокола с параметром *WSAEnumProtocols* и

структурой *WSAPROTOCOLINFO* содержится в главе 5.) Для создания сокета вам придется самостоятельно пометить флаг *SOCK\_RAW*. Приведенный далее отрывок кода поясняет создание простого сокета с использованием ICMP в качестве базового IP-протокола.

```
SOCKET s;

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
// Или
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    // При создании сокета возвращена ошибка
}
```

Поскольку простые сокеты позволяют манипулировать базовым транспортом, их можно использовать для причинения вреда. Поэтому в Windows NT их вправе создавать только члены группы Administrators. Windows 95 и Windows 98 не налагают никаких ограничений.

Чтобы обойти систему безопасности Windows NT, блокируйте проверку защиты на простых сокетах, создав следующую переменную системного реестра и присвоив ей 1 как значение типа *DWORD*.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
\Services\Afd\Parameters\DisableRawSecurity
```

После изменения системного реестра перезагрузите компьютер.

В этом примере кода мы использовали протокол ICMP, но вы можете также применить IGMP, UDP, IP или упрощенный IP, задействовав флаги *IPPROTO\_IGMP*, *IPPROTO\_JDP*, *IPPROTO\_IP* или *IPPROTO\_RAW* соответственно. Однако есть ограничение: в Windows NT 4, Windows 98 и Windows 95 (с Winsock 2) при создании простых сокетов допустимо применение только IGMP и ICMP. Флаги протокола *IPPROTO\_JDP*, *IPPROTO\_IP* и *IPPROTO\_RAW* требуют использования параметра сокета *IPHDRINCL*, не поддерживаемого этими платформами. Windows 2000, однако, поддерживает этот параметр, а потому можно манипулировать самим заголовком IP (*IPPROTO\_IP*), заголовком TCP (*IPPROTO\_TCP*) и заголовком UDP (*IPPROTO\_UDP*).

Создав простой сокет с соответствующими флагами протокола, используйте его описатель в вызовах передачи и приема данных. При создании простых сокетов заголовок IP будет добавляться в данные, возвращаемые после приема каждой порции информации, независимо от того, задан ли параметр *IPHDRINCL*.

## Протокол ICMP

ICMP — инструмент передачи сообщений между узлами. Большинство сообщений ICMP касается ошибок взаимодействия узлов, остальные — применяются для опроса узлов. Этот протокол использует IP-адресацию, поскольку

встроен в дейтаграмму IP. Поля сообщений ICMP показаны на рис 13-1. Сообщение ICMP заключено в заголовок IP.

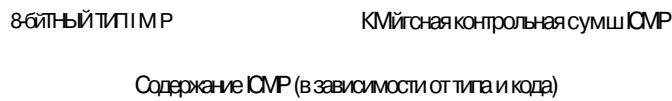


Рис. 13-1. Заголовок ICMP

В первом поле указан тип сообщения ICMP, которое может быть классифицировано или как запрос, или как ошибка. Поле кода определяет тип запроса или сообщения. В поле контрольной суммы заключена 16-битная комплементарная сумма заголовка ICMP. Наконец, его содержание зависит от типа ICMP и кода (табл. 13-1)

Табл. 13-1. Типы сообщения ICMP

Тип	Запрос или тип ошибки	Код	Описание
0	Запрос	0	Эхо-ответ
3	Ошибка адресат недоступим	0	Сеть недоступна
		1	Узел недоступен
		2	Протокол недоступен
		3	Порт недоступен
		4	Необходима фрагментация, но задан бит ее запрета
		5	Ошибка на исходном маршруте
		6	Сеть адресата неизвестна
		7	Узел адресата неизвестен
		8	Исходный узел изолирован (устаревшее)
		9	Сеть адресата административно изолирована
		10	Узел адресата административно изолирован
		11	Сеть недоступна для TOS
		12	Узел недоступен для TOS
		13	Связь административно запрещена фильтрацией
		14	Нарушение приоритета узлов
		15	Пренебрежение приоритетом узла
4	Ошибка	0	Подавление отправителя
5	Ошибка: перенаправление	0	Перенаправление для сети
		1	Перенаправление на узел
		2	Перенаправление на TOS и сеть
		3	Перенаправление на TOS и узел
8	Запрос	0	Эхо-запрос
9	Запрос	0	Оповещение маршрутизатора
10	Запрос	0	Запрос маршрутизатора

Табл. 13-1. (продолжение)

Тип	Запрос или тип ошибки	Код	Описание
11	Ошибка: время превышено	0	TTL в ходе транзита равен 0
		1	TTL в ходе повторной сборки р>ав««вмтомП.
12	Ошибка-проблема параметра	0	Неверный заголовок IP
	❄* МП)	1	* Отсутствует требуемый параметр
13	Запрос	0	Запрос штампа времени г1 е,иЧ *
14	Запрос	0	Ответ штампа времени
15	Запрос	0	Информационный запрос 180(1
16	Запрос "i <	0	Информационный ответ
17	Запрос ч*	0	Маска адреса запроса ,,
18	Запрос	0	Маска адреса ответа

Сгенерированное сообщение ICMP об ошибке всегда содержит заголовок IP и первые 8 байт дейтаграммы IP, ставшей причиной ошибки. Это позволяет узлу, принявшему сообщение об ошибке, связать его с конкретным протоколом и процессом. В нашем случае Ping полагается на эхо-запрос и эхо-ответ ICMP, а не на сообщение об ошибке. Применение ICMP в основном ограничено реакцией узлов на проблемы TCP или UDP В следующем разделе мы рассмотрим, как с помощью протокола ICMP с простым сокетом создать запрос Ping, используя эхо-запрос и сообщения эхо-ответа.

Пример Ping

'aTebwjt

Ping часто используют, чтобы определить, является ли конкретный узел действующим и доступным по сети. Создав эхо-запрос ICMP и направив его на интересующий узел, можно определить, доступен ли этот узел. Конечно, доступность узла не гарантирует пользователю сокета возможность соединиться с процессом на этом узле (поскольку, например, процесс на удаленном сервере может быть не настроен для соединения). Но это означает, что сетевой уровень удаленного узла реагирует на сетевые события. По сути, в этом примере Ping исполняет следующие шаги.

1. Создает сокет типа *SOCKJIAW* под протокол *IPPROTOICMP*.
2. Создает и инициализирует заголовок ICMP.
3. Вызывает *sendto* или *WSASENDTO*, чтобы отправить запрос ICMP на удаленный узел.
4. Вызывает *recvfrom* или *WSARECVFROM* для приема любых ICMP-откликов.

Когда вы отправляете эхо-запрос ICMP, удаленный компьютер прерывает его и создает сообщение эхо-ответа. Если по неким причинам узел не доступен, соответствующее ICMP-сообщение об ошибке (например, «узел адресата не доступен») будет возвращено маршрутизатором где-нибудь на пути к месту назначения. Если физически сетевое соединение с узлом имеется, но удаленный узел временно не доступен или не реагирует на сетевые события,

вам придется задать время ожидания, чтобы определить это. Листинг 13-1 на примере Ping с поясняет, как создать сокет для отправки и приема ICMP-пакетов, а также как использовать параметр сокета *JPTIONS* для реализации функции записи маршрута

116

1 \*

### Листинг 13-1. Пример Ping

```
// Имя модуля Ping с
//
// II Параметры командной строки:
// Ping [host] [packet-size]
//
// host Строковое имя узла адресата
// packet-size Целый размер отправляемого пакета (меньше 1024 байт)
//
#pragma pack(1)
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#define IP_RECORD_ROUTE 0x7
// Структура заголовка IP
typedef struct _iphdr
{
    unsigned int h_len 4; // Длина заголовка
    unsigned int version:4; // Версия IP
    unsigned char tos; // Тип службы
    unsigned short total_len; // Полный размер пакета
    unsigned short ident, // Уникальный идентификатор
    unsigned short frag_and_flags; // Флаги
    unsigned char ttl; // Срок жизни
    unsigned char proto; // Протокол (TCP, UDP и т.п.)
    unsigned short checksum; // Контрольная сумма IP
    unsigned int sourceIP;
    unsigned int destIP;
} IpHeader,
#define ICMP_ECHO 8
#define ICMP_ECHOREPLY 0
#define ICMP_MIN 8 // Пакет ICMP не меньше 8 байт (заголовок)
// Структура заголовка ICMP
//
typedef struct _icmphdr
```

Листинг 13-1. (продолжение)

(штжшбоС^) t H nmtat,

```

{
    BYTE i_type;
    BYTE i_code; // Тип субкода
    USHORT i_cksum;
    USHORT i_id;
    USHORT i_seq, 1
    // Это нестандартный заголовок, но мы резервируем место для времени
    ULONG timestamp,
} IcmpHeader, it

axe s\ V.

// Расширенный заголовок IP - применяшшишфаметром \
// сокета IP_OPTIONS • „;$) ;0 « »W*J8J_I it.

typedef struct _ipoptionhdr '
{
    // Тип параметра
    unsigned char code;
    // Длина расширенного
    unsigned char : len; up
    // Смещение на параметры
    unsigned char ptr;
    // Перечень IP-адресов '
    unsigned long addr[9];
} IpOptionHeader;

#define DEF_PACKET_SIZE 32 // Стандартный размер пакета \
#define MAX_PACKET 1024 // Максимальный размер ICMP-пакета » .PNHXHV* \
#define MAX_IP_HDR_SIZE 60 // Максимальный размер заголовка IP \
// с параметрами ewr-wifl \

BOOL bRecordRoute; шм «yt И т>влэ(» М \
int datasize; о »^9ф¥¹ 'чьаевху япд \
char «Ipdest; \

(«Ua tnt

//
// Функция: usage w, t»t
//
// Описание: (Г < asie) ellrtw
// Выводит информацию об использовании
//
void usage(char *progname)
{
    pnntf("usage` ping -r <host> [data size]\n");
    printf(" -r record route\n");
    printf(" host remote machine to");
    printfC" datasize can be up to 1 KB\n");
    ExitProcess(-i); tttxo *

    »+
    0)

// Функция: FillICMPData

```



## Листинг 13-1. (продолжение)

ЭКИ,

// Описание:

r \

// Вспомогательная функция для заполнения полей нашего запроса

void FillICMPData(char \*icmp\_data, mt datasize) ;|.

IcmpHeader \*icmp\_hdr = NULL;

char «datapart = NULL; Outr

icmp\_hdr = (IcmpHeader\*)icmp\_data;

\*I \

icmp\_hdr-&gt;i\_type = ICMP\_ECHO; // Эхо-запрос ICMP

icmp\_hdr-&gt;i\_code = 0;

\

icmp\_hdr-&gt;i\_id = (USHORT)GetCurrentProcessId();

o V

icmp\_hdr-&gt;i\_cksum = 0;

; V

icmp\_hdr-&gt;i\_seq = 0;

\%

TO.

wifY

datapart = icmp\_data + sizeof(IcmpHeader);

&gt;

// Поместите какие-нибудь данные в буфер

memset(datapart,'E', datasize - sizeof(IcmpHeader));

// Функция: checksum

\*etv

//

"eb,

// Описание:

// Вычисляет 16-битную комплементарную сумму

oo»Яa J0 "

// для указанного буфера с заголовком

.eeteb i'u

USHORT checksum(USHORT «buffer, int size)

unsigned long cksum=0;

ajteu :u<sup>k</sup>.

&lt; "ч

while (size &gt; 1)

, :SN-

' i'

cksum += «buffer++;

\<sup>4</sup>

size -= sizeof(USHORT);

-лц

if (size)

iq

{

i •

cksum += \*(UCHAR\*)buffer;

cksum = (cksum » 16) + (cksum &amp; 0xffff);

cksum += (cksum » 16);

return (USHORT)Ccksum);

// функция: DecodeIPOptions

Листинг 13-1. (продолжение)

Л С Г

```
// Описание: . 'o1T3л; **
// При наличии расширенного заголовка находит параметры IP
// в нем и выводит значения параметра записи маршрута ткм ,,Л

void DecodeIPOptions(char *buf, int bytes)

    IpOptionHeader «ipopt = NULL;
    NADDR maddr; , * «. ttfc-q
    int i; , k< r
    HOSTENT «host = NULL;

    ipopt = (IpOptionHeader *)(buf + 20);

    printf("RR: ");
    for(i = 0; i < (ipopt->ptr / 4) - 1; i++) ei is* & г ,.^и

        inaddr.S_un.S_addr = ipopt->addr[i];
        if (i != 0)
            printf(" ");
        host = gethostbyaddr((char *)&inaddr.S_un.S_addr,
            sizeof(inaddr.S_un.S_addr), AF_INET); Jicwwi) <t
        if (host)
            printf("(X-15s) Xs\n", inet_ntoa(inaddr), host->h_name); jrnt<j
        else
            printf("(X-15s)\n", inet_ntoa(inaddr)); , oiut^n

    return; ;(p3з, jk< JSQ: , " ,bt »

// Функция: DecodelCMPHeader
// nitJei
// Описание: {
// II Ответом является пакет IP. Следует декодировать заголовок IP
// для нахождения данных ICMP ч^ ^XeV Ытч
// II >
void DecodeICMPHeader(char «buf, int bytes, tni
    struct sockaddr_in *from)
{
    * ! m
    IpHeader .iphdr = NULL;
    IcmpHeader «icmphdr = NULL; ,3Zi? s
    unsigned short iphdrlen;
    DWORD tick;
    static int icmpcount = 0; «

    iphdr = (IpHeader *)buf;
    // Количество 32-х битных слов помноженное на 4 равно числу байт
```

см. след. стр.



Листинг 13-1. {продолжение}

```

        case 'r': // Параметр записи маршрута
            bRecordRoute = TRUE;
            break;
        default:
            usage(argv[0]);
            break;

    else if (isdigit(argv[i][0]))
        datasize = atoi(argv[i]);
    else
        lpdest = argv[i];

//
// Функция: main

// Описание:
//     Настраивает простой сокет ICMP и создает заголовок ICMP. Добавляет
//     соответствующий расширенный заголовок IP и начинает рассылку
//     эхо-запросов ICMP по конечным точкам. Для каждой отправки и
//     приема мы задаем таймаут во избежание простоя, когда конечная
//     точка не отвечает. Принятый пакет декодируется.

int main(int argc, char **argv)

{
    WSADATA wsaData;
    SOCKET sockRaw = INVALID_SOCKET;
    struct sockaddr_in dest,
        from;
    int fromlen = sizeof(from),
        timeout = 1000,
        ret;
    char *icmp_data = NULL;
    unsigned int addr = 0;
    USHORT seq_no = 0;
    struct hostent *hp = NULL;
    IpOptionHeader ipopt;

    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
}

```

см. след. стр.

Листинг 13-1. *{продолжение}*

г-£МнмтсТ1

```

}
ValidateArgs(argc, argv);

//
// Параметрам SO_RCVTIMEO и SO_SNDTIMEO требуется флаг
// WSA_FLAG_OVERLAPPED. Если последний параметр WSASocket - NULL,
// весь ввод-вывод на сокете синхронный, внутренний код ожидания
// пользовательского режима никогда не будет выполнен, а потому {
// окончательно блокируется ввод-вывод в режиме ядра.
// Для сокета, созданного функцией socket, атрибут перекрытого
// ввода-вывода задается автоматически. Однако в данном случае
// создания простого сокета нужно использовать WSASocket.
//
// Если вы хотите использовать таймаут с синхронным неперекрывтым {
// сокетом, созданным WSASocket с последним параметром
// равным NULL, задайте таймаут функцией select либо задействуйте \\
// WSAEventSelect и укажите таймаут в функции ^чдкн^Ф \\
// WSAWaitForMultipleEvents. \\
// аэмтО \\
sockRaw = WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0, _H \\
WSA_FLAG_OVERLAPPED); _O \\
if (sockRaw == INVALID_SOCKET) <_B \\
{ >qr. \\
    printf("WSASocket () failed: %d\n", WSAGetLastError()); _Y \\
    return -1; _\\
}

if (bRecordRoute) *

// Настройка отправки расширенного заголовка
// IP с каждым ICMP-пакетом fMoee 1-

ZeroMemory(&iptopt, sizeof(iptopt));
iptopt.code = IP_RECORD_ROUTE; // Параметр записи маршрута
iptopt.ptr = 4; // Укажите смещение для первого адреса
iptopt.len = 39; // Длина расширенного заголовка

ret = setsockopt(sockRaw, IPPROTO_IP, IP_OPTIONS, (char *)&iptopt, sizeof(iptopt)); i»
if (ret == SOCKET_ERROR) -_s ?ol ben.
< |"
    printf("setsockopt(IP_OPTIONS) failed: %d\n", WSAGetLastError());

// Настройка таймаутов отправки и приема
//
bread = setsockopt(sockRaw, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout, sizeof(timeout));
if (bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_RCVTIMEO) failed: %d\n", WSAGetLastError());
}

```

**Листинг 13-1.** (продолжение)

```

return -1;
>
timeout = 1000;
bread = setsockopt(sockRaw, SOL_SOCKET, SO_SNDTIMEO, (char*)&timeout, sizeof(timeout));
if (bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_SNDTIMEO) failed: %d\n", WSAGetLastError());
    return -1;
}
memset(&dest, 0, sizeof(dest));
//
// При необходимости разрешается имя конечной точки
II
dest.sin_family = AF_INET;
if ((dest.sin_addr.s_addr = inet_addr(lpdest)) == INADDR_NONE)
{
    if ((hp = gethostbyname(lpdest)) != NULL)
    {
        memcpy(&dest.sin_addr, hp->h_addr, hp->h_length);
        dest.sin_family = hp->h_addrtype;
        printf("dest.sin_addr = %s\n", inet_ntoa(dest.sin_addr));
    }
    else
    {
        printf("gethostbyname() failed: %d\n", WSAGetLastError());
        return -1;
    }
}

// Создание ICMP-пакета

datasize += sizeof(IcmpHeader);

icmp.data = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, MAX_PACKET);
recvbuf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, MAX_PACKET);
if (!icmp_data)
{
    printf("HeapAlloc() failed: %d\n", GetLastError());
    return -1;
}

memset(icmp_data, 0, MAX_PACKET);
FillICMPData(icmp_data, datasize);

// Начало отправки/приема ICMP-пакетов

while(d)

```

Листинг 13-1.    {продолжение}

```

static int nCount = 0;
int        bwrote,

if (nCount++ == 4)
    break,

((icmpHeader*)icmp_data)->i_cksum = 0;
((icmpHeader*)icmp_data)->timestamp = GetTickCount();
((icmpHeader*)icmp_data)->i_seq = seq_no++;
((icmpHeader*)icmp_data)->i_cksum =
    checksum((USHORT*)icmp_data, datasize);

bwrote = sendto(sockRaw, icmp_data, datasize, 0, (struct sockaddr*)&dest, sizeof(dest));
if (bwrote == SOCKET_ERROR)
{
    if (WSAGetLastError() == WSAETIMEDOUT)
    {
        printf('timed out\n');
        continue;
    }
    printf('sendto failed- %d\n', WSAGetLastError());
    return -1,

if (bwrote < datasize)
{
    printf("Wrote %d bytes\n", bwrote);
}
bread = recvfrom(sockRaw, recvbuf, MAX_PACKET, 0, (struct sockaddr*)&from,
&fromlen);
if (bread == SOCKET_ERROR)
{
    if (WSAGetLastError() == WSAETIMEDOUT)
    {
        printf("timed out\n");
        continue;
    }
    printf("recvfrom() failed' %d\n", WSAGetLastError()); b.gets-.') >i
    return -1;
}
DecodeICMPHeader(recvbuf, bread, &from);

Sleep(1000),

}
// Очистка
//
if (sockRaw != INVALID_SOCKET)
    closesocket(sockRaw),
HeapFree(GetProcessHeap(), 0, recvbuf);
HeapFree(GetProcessHeap(), 0, icmp_data),

```

**Листинг 13-1.** *{продолжение}*

```
WSACleanup();  
return 0;
```

Одна из значимых особенностей этого примера Ping — использование параметра сокета *IP\_OPTIONS*. Мы применяем параметр записи маршрута IP, чтобы когда ICMP-пакет достигнет маршрутизатора, его IP-адрес был добавлен в расширенный заголовок IP в месте, обозначенном полем смещения. Это смещение увеличивается на 4 каждый раз, когда маршрутизатор добавляет адрес, поскольку для IP версии 4 длина IP-адреса равна 4 байтам. В этой книге мы не касаемся IP версии 6, поскольку имеющиеся платформы Windows пока не поддерживают ее. После приема эхо-ответа *расширенный заголовок* (option header) декодируется и на экран выводятся IP-адреса и имена узлов пройденных маршрутизаторов. (Подробнее о других типах доступных параметров IP — в главе 9)

## Программа Traceroute

Это еще один ценный сетевой инструмент, позволяющий определять IP-адреса маршрутизаторов, проходимых запросом на пути к определенному узлу. Ping также позволяет определить IP-адреса маршрутизаторов, используя параметр записи маршрута в расширенном заголовке IP, однако его возможности ограничены всего девятью транзитами — максимальным пространством, предназначенным для адресов в расширенном заголовке. Транзит происходит всякий раз, когда IP-дейтаграмма проходит маршрутизатор для перехода в другую физическую сеть. Для маршрутов с более чем девятью такими переходами используйте Traceroute.

В основе Traceroute лежит идея отправки UDP-пакета адресату и постепенного изменения времени *жизни* (time-to-live, TTL). Первоначально TTL пакета равен 1, и когда пакет достигает первого маршрутизатора, его TTL сбрасывается, и маршрутизатор генерирует ICMP-пакет со сведениями о превышении лимита времени. Тогда начальное значение TTL увеличивается на 1, так что на сей раз UDP-пакет достигает следующего маршрутизатора, а тот тоже отправляет ICMP-пакет по превышению лимита времени. Совокупность этих ICMP-сообщений дает список IP-адресов, пройденных на пути к конечной точке. Когда TTL увеличится настолько, что UDP-пакет достигнет исковой конечной точки, возвращается ICMP-сообщение о недостижимости порта, поскольку на получателе ни один процесс не ждет вашего сообщения.

Traceroute полезна тем, что дает подробнейшую информацию о маршруте до конкретного узла — это часто необходимо при многоадресной передаче или проблемах с маршрутизацией. Впрочем, программная реализация функциональности Traceroute необходима реже, чем Ping.

Существует два способа выполнить программу Traceroute. Во-первых, вы можете использовать UDP-пакеты и отправлять дейтаграммы, постепенно изменяя TTL. Каждый раз по истечении TTL будет возвращаться ICMP-сообщение



ние. Этот метод требует наличия обычного сокета протокола UDP (см. главу 7) для отправки сообщений, а также сокета типа *SOCK\_RAW* под протокол *IPPROTOICMP* для чтения возвращенных сообщений. TTL для UDP-сокета можно контролировать через параметр сокета *IPJTTL*, либо создать UDP-сокет и использовать параметр *IP\_HDRINCL* (см. далее в этой главе), чтобы вручну задать TTL в заголовке IP. Но это чересчур трудоемкая задача.

Другой метод — просто отправлять ICMP-пакеты адресату с постепенным изменением TTL. В результате также возвращаются ICMP-сообщения об ошибке по истечении TTL. Это похоже на пример Ping, в том смысле, что требуется только один сокет (для протокола ICMP). В папке с примерами на прилагаемом компакт-диске есть пример реализации Traceroute с использованием ICMP-пакетов (Traceroute.c).

Протокол IGMP

Протокол IGMP используется многоадресным вещанием IP для управления членством в многоадресных группах. (Об использовании Winsock для вступления в такую группу и выхода из нее — в главе 11.) Когда программа соединяется с многоадресной группой, она отправляет ЮМР-сообщение каждому маршрутизатору локальной сети, используя специальный адрес группы всех маршрутизаторов — 224.0.0.2. Сообщение информирует маршрутизаторы о наличии получателя для любых данных, предназначенных для многоадресной группы. Только после этого маршрутизаторы начинают перенаправлять данные группы зарегистрированному получателю. Без этой возможности многоадресная передача данных ничем не отличалась бы от широковещания.

Некоторую путаницу создают две версии протокола IGMP: версии 1 и 2, которые описаны в RFC 1112 и RFC 2236 соответственно. Главное различие между ними — версия 1 не позволяет узлу дать маршрутизатору указание о прекращении отправления данных, предназначенных для многоадресной группы. Другими словами, когда узел решает выйти из группы, он не уведомляет об этом маршрутизатор, который продолжает распространять данные для этой группы, пока не убеждается в отсутствии отклика. Версия 2 предусматривает четкое и немедленное уведомление узлами маршрутизаторов о выходе из группы. Кроме того, немного различаются форматы заголовка двух версий (рис. 13-2 и 13-3)- Оба заголовка просты, поскольку имеют размер всего 8 байт.

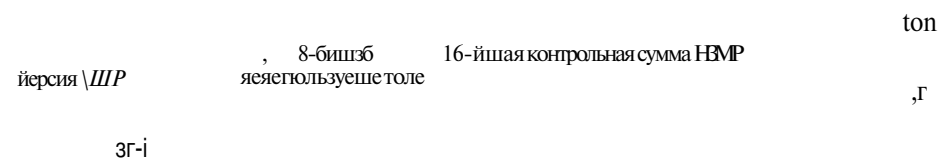
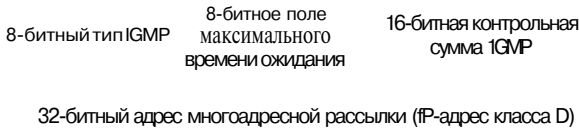


Рис. 13-2. Формат заголовка IGMPv1



**Рис. 13-3. Формат заголовка IGMPv2**

В версии 1 два четырехбитных поля: первое — версия IGMP, второе — тип сообщения. В версии 2 эти два поля заменяет одно восьмибитное. Неиспользуемое в версии 1 поле становится полем *максимального времени ожидания* (Max Response Time). Оно используется только при передаче запроса о членстве: определяет максимальное время ожидания ответа в десятых долях секунды. Во всех других случаях отправитель задает в этом поле 0, и получатели игнорируют его.

Для IGMPv1 полем версии всегда является первое, а в поле типа задается одно из двух возможных значений- 0x1 — запрос о членстве в группе, 0x2 — отчет о членстве. Маршрутизаторы используют запрос о членстве узла (0x1), чтобы определить, к какой многоадресной группе причислен узел. В этом случае полю адреса группы присваивается 0. Маршрутизатор отправляет пакет по адресу всех узлов (224.0.0.1)

Если узлы по-прежнему являются членами какой-либо группы, каждый из них возвращает отчет о членстве (тип ЮМР-сообщения 0x2) вместе с адресом группы, в которой зарегистрирован. При первичной регистрации узла в группе сообщение с отчетом о членстве также отправляется на групповой адрес маршрутизаторов.

В версии 2 протокола IGMP предусмотрены следующие типы сообщений.

**III 0x11** — запрос о членстве;

• **0x12** — отчет о членстве версии 1;

**В 0x16** — отчет о членстве версии 2;

; j ^

**III 0x17** — выход из группы.

^ „ „

.

Как видите, добавлены два новых типа сообщения: отчет версии 2 (0x16) и выход из группы (0x17). Два других сообщения аналогичны сообщениям версии 1 (запрос и отчет по членству), хотя их значения различаются. Однако не забывайте, что IGMP-пакет версии 2 содержит поля версии и типа сообщения в одном поле, а 0x11 в двоичном исчислении — 00010001, что похоже на пакет IGMP со значением 1 в поле версии и типа.

Запрос о членстве (0x11) в версии 2 слегка отличается от такого запроса в версии 1, поскольку позволяет выяснить членство для конкретного группового адреса.

Механизм таков: группе отправляется пакет запроса о членстве с конкретным адресом в поле адреса. Кроме того, поле максимального времени ожидания позволяет задать время, в течение которого узлы должны отвечать на запрос до прекращения перенаправления им многоадресного трафика указанной группы.

## Использование *IP\_HDRINCL*

Как уже упоминалось, с простыми сокетами могут работать только некоторые протоколы типа ICMP и IGMP. Нельзя создать простой сокет при помощи *IPPROTOJDP* и манипулировать заголовком UDP, это же относится к TCP. Для манипулирования заголовком IP, TCP или UDP (да и любого другого вложенного в IP протокола) необходимо использовать с простым сокетом параметр *IP\_HDRINCL*, который позволяет сформировать собственный заголовок IP или других протоколов.

Если вы хотите реализовать собственную схему протокола вложенного в IP, создайте простой сокет и используйте в качестве ссылки на протокол значение `IPPROTO_RAW`. Это позволит самостоятельно задать поле протокола в заголовке IP и сформировать заголовок протокола. В этом разделе мы по шагам рассмотрим формирование собственных пакетов UDP. Разобравшись, как манипулировать заголовком UDP, вы без труда сможете создать собственные заголовки протокола и управлять другими протоколами под IP.

При использовании параметра *IPHDRINCL* требуется самостоятельно заполнять заголовок IP для каждого вызова отправки, а также заголовки любых других вложенных протоколов (см главу 9, рис 9-3) Заголовок UDP проще Его величина всего 8 байт, при том он содержит всего четыре поля (рис 13-4) Первые два хранят 16-битные номера портов отправителя и адресата Третье поле определяет длину UDP, равную суммарной длине заголовка UDP и данных (в байтах) Четвертое — поле контрольной суммы, которое мы рассмотрим вкратце В конце пакета UDP размещаются данные

11-битный якор отправителя	
16-битная длина аШР	16-битная контрольная сумма UDP

**Рис. 13-4. Формат заголовка UDP**

Поскольку UDP — ненадежный протокол, вычисление контрольной суммы не обязательно, однако, для полноты картины мы рассмотрим и этот аспект. В отличие от контрольной суммы IP, которая охватывает только заголовок IP, контрольная сумма UDP включает данные и часть заголовка IP. Дополнительные поля, требуемые для вычисления контрольной суммы UDP, получили название псевдозаголовка. Он состоит из следующих элементов:

- В 32-битный IP-адрес отправителя (заголовок IP), эп

III 32-битный IP-адрес получателя (заголовок IP),

*III* 8-битное обнуленное поле,

- 8-битное поле протокола,
- 16-битная длина UDP

Помимо этих элементов существуют заголовок UDP и данные. Метод вычисления контрольной суммы аналогичен IP и ICMP. 16-битная комплементарная сумма. Если данные выражены нечетным числом, для вычисления конт-

рольной суммы их нужно дополнить нулевым байтом (в конце) Это дополнительное поле не передается вместе с остальными данными Поля, необходимые для вычисления контрольной суммы показаны на рис 13-5 Первые три 32-битных слова составляют псевдозаголовок UDP, затем следуют заголовок UDP и данные Заметьте так как контрольная сумма рассчитана по 16-битным значениям, может потребоваться дополнить данные нулевым байтом

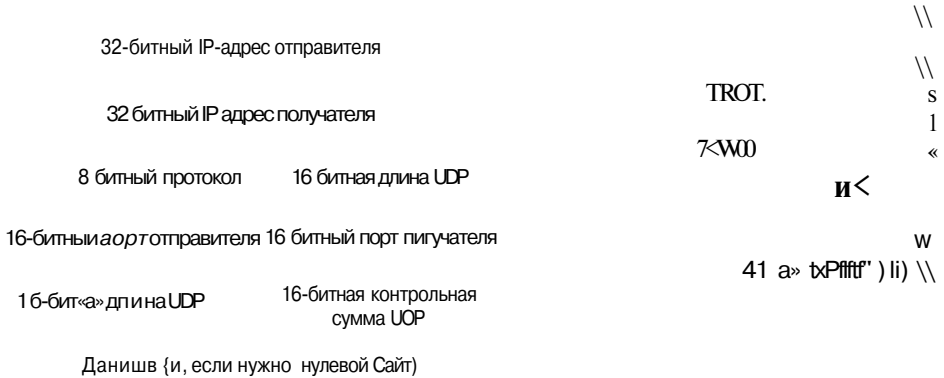


Рис. 13-5. Псевдозаголовок UDP

Программа, взятая нами в качестве примера (см листинг 13-2), просто отправляет пакет UDP любому адресату IP с любого исходного адреса IP и выбранного порта Во-первых, необходимо создать простой сокет с флагом *IP\_HDRINCL*

```

SOCKET s;
int ret;

s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,
WSA_FLAG_OVERLAPPED, 0, (char *)0, sizeof(bOpt)),
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)0, sizeof(bOpt)),

```

Заметьте, что мы создали простой сокет с использованием протокола *IPPROTO\_UDP* Поскольку это требует поддержки параметра *IP\_HDRINCL*, в качестве платформы подходит только Windows 2000 Далее на примере *Iphdnn* с поясняется, как использовать простые сокеты и параметр *IP\_HDRINCL* для манипулирования заголовками IP и UDP на исходящих пакетах

#### Листинг 13-2. Пример простого UDP-сокета

```

// Имя модуля Iphdrinc
#pragma pack(T)

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>

```

см след стр

Табл. 13-2.      *(продолжение)*

```

((include <stdio.h>
«include <stdlib.h>

(define MAX.MESSAGE        4068
«define MAX_PACKET        4096
//
// Настройка некоторых значений по умолчанию

«define DEFAULT.PORT        5150
«define DEFAULT.IP         "10.0.0.1"
«define DEFAULT_COUNT      5
«define DEFAULT.MESSAGE    "This is a test"

// Определение IP-заголовка. Поля версии и длины заданы одним
// символом, поскольку мы не можем объявить два 4-битных поля,
// чтобы компилятор не выравнивал их хотя бы по границе одного байта.

typedef struct ip_hdr

    unsigned char   ip_verlen;        // Версия и длина IP
    unsigned char   ip_tos;          // Тип службы IP
    unsigned short   ip_totallength;   // Полная длина
    unsigned short   ip_id;          // Уникальный идентификатор
    unsigned short   ip_offset;      // Поле смещения фрагмента
    unsigned char   ip_ttl;          // Срок жизни
    unsigned char   ip_protocol;     // Протокол (TCP, UDP и т.п.
    unsigned short   ip_checksum;     // Контрольная сумма IP
    unsigned int    ip_srcaddr;      // Исходный адрес
    unsigned int    ip_destaddr;     // Целевой адрес
} IP_HDR, »PIP_HDR, FAR» LPIP_HDR;

// Определение заголовка UDP
//
typedef struct udp_hdr
{
    unsigned short   src_portno;      // Номер порта отправителя
    unsigned short   dst_portno;      // Номер порта получателя
    unsigned short   udp_length;      // Длина пакета UDP
    unsigned short   udp_checksum;    // Контрольная сумма UDP (необяз.)
} UDP_HDR, .PUDP_HDR;

// Глобальные переменные
//
unsigned long   dwToIP,              // IP-адрес получателя
             dwFromIP;              // IP-адрес отправителя (фиктивный)
unsigned short   iToPort,            // Порт получателя
             iFromPort;            // Порт отправителя (фиктивный)

```

Табл. 13-2. (продолжение)

**WORD** dwCount; // Количество повторов отправки  
char strMessage[MAX\_MESSAGE]; // Отправляемое сообщение

```
// Описание:
// Вывод информации об использовании и выход wb
//                                     :*-
void usage(char «programe»                                     >>
{
    printf("usage: Xs [-fp:int] [-fi:str] [-tp:int] [-ti:str]\n"
           [-n:int] [-m:str]\n", programe);
    printf("    -fp:int    From (sender) port number\n"); id
    printf("    -fi:IP     From (sender) IP address\n");
    printf("    -fp:int    To (recipient) port number\n"); «8
    printf("    -fi:IP     To (recipient) IP address\n"); }
    printf("    -n:int    Number of times to read messaged");
    printf("    -m:str    Size of buffer to read\n\n");
    ExitProcess(i);          *» • no
}                             -vi

// Функция: ValidateArgs                                     41

// Описание:                                     r r
// II Анализирует параметры командной строки и задает некоторые
// глобальные флаги согласно планируемым действиям

void ValidateArgs(int argc, char «*argv»
{
    int i;                                     ; It

    IToPort = DEFAULT_PORT;
    IFromPort = DEFAULT_PORT;
    dwToIP = inet_addr(DEFAULT_IP); C -18) \t
    dwFromIP = inet_addr(DEFAULT_IP);
    dwCount = DEFAULT_COUNT;
    strcpy(strMessage, DEFAULT_MESSAGE); H

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'f: // Исходный адрес
                {
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':

```

см. след. стр.

Табл. 13-2.
(продолжение)
м'^\,

```

        if (strlen(argv[i]) > 4)
M*      iFromPort = atoi(&argv[i][4]);
        break;
        case 'i':
            if (strlen(argv[1]) > 4) :eNMS»w
                dwFromIP = inet_addr(&argv[i][4]); ф*и аовчй
            break;
        default:
            usage(argv[0]);
            break;
    }
    break;
case 'f:      // Целевой адрес
    switch (tolower(argv[i][2]))

        case 'p':
            if (strlen(argv[i]) > 4)
                iToPort = atoi(&argv[i][4]);
            break;
        case 'i':
            if (strlen(argv[i]) > 4) V
                dwToIP = inet_addr(&argv[i][4]); uxwft \\\
            break;
            default:
                &5чп0 \s
            usage(argv[0]);
            break;
        >
        break;
        case 'n':      // Число повторов отправки сообщения
            if (strlen(argv[i]) > 3) tul
                dwCount = atol(&argv[i][3]);
            break;
        case 'm':
            if (strlen(argv[i]) > 3)
                strcpy(strMessage, &argv[i][3]);
            break;
        default:
            -j_t
            usage(argv[0]);
            break;
            , I * 2)->aT

i      " .      (      (0      n»)) It

T0tU rPI      'Vtt*'      '> Po'      J1ш#

//
// Функция: checksum
//
// Описание:

```

Табл. 13-2. (продолжение)

// Вычисляет 16-битную контрольную сумму  
 // для указанного буфера

```
USHORT checksum(USHORT buffer,
```

```
    unsigned long cksum=0;
```

```
    while (size > 1)
```

```
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
```

```
    if (size)
```

```
    {
        cksum += (USHORT)buffer;
```

```
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
```

```
    return (USHORT)(cksum);
```

```
}
```

```
//
```

```
// Функция: main
```

```
//
```

```
// Описание:
```

```
// Анализирует параметры командной строки и загружает Wmsock.
```

```
// Создает простой сокет и включает параметр IP_HDRINCL.
```

```
// Затем собирает заголовки пакетов IP и UDP путем
```

```
// задания правильных значений и вычисления контрольных сумм.
```

```
// После этого вписывает данные и высылает адресату.
```

```
int main(int argc, char **argv)
```

```
{
```

```
    WSADATA wsd;
```

```
    SOCKET s;
```

```
    BOOL bOpt;
```

```
    struct sockaddr_in remote; // Структуры IP-адресации
```

```
    IP_HDR lpHdr;
```

```
    UDP_HDR udpHdr; +
```

```
    int ret;
```

```
    int i;
```

```
    unsigned short lTotalSize, // Для заполнения разных заголовков
```

```
    iUdpSize, // требуется множество размеров
```

```
    illdpChecksumSize,
```

```
    llPVersion,
```

```
    ilPSize,
```

```
    cksum = 0;
```

```
    char buf[MAX_PACKET],
```

см. след. стр.



Табл. 13-2.    {продолжение)

• \* \*

```

        • ptr = NULL; ,шм      • як    - «.,      <ини>в    \
IN_ADDR      addr;      (Щл      янд    \
                                                    \
// Анализ и вывод параметров командной строки    -jd*      rto      ТЧОН8Ц

ValidateArgs(argc, argv);      {#-      ^«'W
addr.S_un.S_addr = dwFromIP;
printf("From IP: <Xs>\n    Port: Xd\n", inet_ntoa(addr), iFromPort); silri*
addr.S_un.S_addr = dwToIP;      }
printf("To IP: <Xs>\n    Port: Xd\n", inet_ntoa(addr), iToPort); •>
printf("Message: [Xs]\n", strMessage);
printf("Count: Xd\n", dwCount);      {
                                                    U
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)      >

    printf("WSAStartup() failed: Xd\n", GetLastErrorO);      {
    return -1;      o

// Создание простого сокета
//
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,0);      '1
if (s == INVALID_SOCKET)      {
    printf("WSASocket() failed: Xd\n", WSAGetLastErrorO);      \
    return -1;      V
>      . Ю \
                                                    \
// Разрешение заголовку содержать параметр      >ooO    \
                                                    >ТВ€    "

bOpt = TRUE;      \
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt, sizeof(bOpt)); \
if (ret == SOCKET_ERROR)      ;. л    t Jn

    printf("setsockopt(IP_HDRINCL) failed: Xd\n", WSAGetLastError(W;    I
    return -1;      Ѱ
}      I
// Инициализация заголовка IP      < I jit*т>e*SO .    j
//      "    I
iTotalSize = sizeof(ipHdr) + sizeof(udpHdr) + strlen(strH*tS«e*)! Я0«_ч*1
                                                    tnJE
ilPVersion=4;      ,    ОЯОЛ
ilPSize = sizeof(ipHdr) / sizeof(unsigned long);      Jiorie

// Версия IP заключена в старших 4 битах ip_verlen.
// Длина заголовка IP (в 32-битных словах) - в младших 4 битах.

ipHdr.ip_verlen = (ilPVersion << 4) | ilPSize;
ipHdr.ip_tos = 0;      // Тип службы IP
ipHdr.ip_totallength = htons(iTotalSize); // Полная длина пакета
    
```

Табл. 13-2. (продолжение)

```

ipHdr.ip_id = 0;           // Уникальный идентификатор: приравнять 0
ipHdr.ip_offset = 0;       // Поле смещения фрагмента i
ipHdr.ip_ttl = 128;        // Срок жизни
ipHdr.ip_protocol = 0x11;  // Протокол (UDP) , rlv
ipHdr.ip_checksum = 0;     // Контрольная сумма IP jo
ipHdr.ip_srcaddr = dwFromIP; // Адрес отправителя k>, . . ,
ipHdr.ip_destaddr = dwToIP; // Адрес получателя

//
// Инициализация заголовка UDP ibs*1.
// V
iUdpSize = sizeof(udpHdr) + strlen(strMessage); // \
udpHdr.src.portno = htons(iFromPort) ;
udpHdr.dst.portno = htons(iToPort) ; -
udpHdr.udp_length = htons(iUdpSize) ; д
udpHdr.udp_checksum = 0 ; j-q

// b*
// Формирование псевдозаголовка UDP для вычисления ,&{**
// контрольной суммы UDP. Псевдо-заголовок состоит из 32-битного кчг
// IP-адреса отправителя, 32-битного IP-адреса получателя,
// нулевого байта, 8-битного поля IP-протокола, 16-битной длины UDP \,
// и самого заголовка UDP вместе с данными ,\
// (дополненными нулем, если их длина нечетная). щ< \
II \
iUdpChecksumSize = 0; \
ptr = buf; •I'-ne'
ZeroMemory(buf, MAX_PACKET); ,f.-oais<

memcpy(ptr, &ipHdr.ip_srcaddr, sizeof(ipHdr.ip_srcaddr));
ptr += sizeof(ipHdr.ip_srcaddr); ,y » i^j*
iUdpChecksumSize += sizeof(ipHdr.ip_srcaddr); ,

memcpy(ptr, &ipHdr.ip_destaddr, sizeof(ipHdr.ip_destaddr)); >stt
ptr += sizeof(ipHdr.ip_destaddr); «*• j&tp'Ti"
iUdpChecksumSize += sizeof(ipHdr.ip_destaddr); }

ptr++;
iUdpChecksumSize += 1;

memcpy(ptr, &ipHdr.ip_protocol, sizeof(ipHdr.ip_protocol));
ptr += sizeof(ipHdr.ip_protocol);
iUdpChecksumSize += sizeof(ipHdr.ip_protocol); ^

memcpy(ptr, &udpHdr.udp_length, sizeof(udpHdr.udp_length)); ! <t
ptr += sizeof(udpHdr.udp_length);
iUdpChecksumSize += sizeof(udpHdr.udp_length);

memcpy(ptr, &udpHdr, sizeof(udpHdr));

```

см. след. стр.

Табл. 13-2. {продолжение)

```

ptr += sizeof(udpHdr);          < K,   *«Л'
iUdpChecksumSize += sizeof(udpHdr); wft          ;l          U
                                     t
for(i = 0; i < strlen(strMessage); i++, ptr++)    ;ТхО -    ,t
    *ptr = strMessage[i];                        d
iUdpChecksumSize += strlen(strMessage);          ;t

cksum = checksum((USHORT *)buf, iUdpChecksumSize);          Л
udpHdr.udp_checksum = cksum;                               ИФНШ \
// '                                     \
// Теперь соберем заголовки IP и UDP с данными,      ihjbu}^1 - ex£24b>U£
// чтобы отправить их                                'rf " Off' "j
// < * » ,                                           ч)
ZeroMemory(buf, MAX_PACKET);                          t
ptr = buf;                                             iu

memcpy(ptr, &ipHdr, sizeof(ipHdr)); ptr += sizeof(ipHdr);    \
memcpy(ptr, &udpHdr, sizeof(udpHdr)); ptr += sizeof(udpHdr); члвоцммэд* \
memcpy(ptr, strMessage, strlen(strMessage));                itsyt \
                                                              "Я \
II Очевидно, что эта структура SOCKADDR_IN не играет роли:  h \
// подходит любой IP-адрес, введенный в качестве адреса получателя м \
// в заголовок IP. Конкретный адресат в поле remote будет т т т т щ) \
// проигнорирован.                                         \
//                                                         ;0
remote.sin_family = AF_INET;
remote.sin_port = htons(iToPort);                        ;{ТЗЖ)А<? ХАМ ,U
remote.sin_addr.s_addr = dwToIP;                          ?

for(i = 0; i < dwCount; i++)

    ret = sendto(s, buf, iTotalSize, 0, (SOCKADDR *)&remote,
        sizeof(remote));
    if (ret == SOCKET_ERROR)

        printf("sendto() failed: Xd\n", WSAGetLastError());
        break;

    else " ^
        printf("sent Xd bytes\n", ret);

closesocket(s) ;                                          i>@<-
WSACleanupQ ;

return 0;

```

После создания сокета с параметром *IPJiiDRINCL*, код начинает заполнять заголовок IP, объявленный как структура *IP\_HDR*. Заметьте: первые два четы-

рехбитных поля объединены в одно, потому что компилятор может выравнивать поля только на границе минимум в 1 байт. В связи с этим, версия IP должна быть указана в старших четырех битах. Поле *ipjprotocol* равно 0x11, что соответствует UDP. Код приравнивает поле *ip\_srcaddr* IP-адресу отправителя (или тому, который вы хотите выдать за исходный), а поле *ip\_destaddr* — IP-адресу получателя. Сетевой стек вычисляет контрольную сумму IP, поэтому она не задается кодом.

Следующий шаг — инициализация заголовка UDP. Это просто, поскольку в нем мало полей. Номера порта отправителя и получателя задаются наряду с размером заголовка UDP. Поле контрольной суммы изначально равно 0. Хотя от вашего кода не требуется вычислять контрольную сумму UDP, в примере это сделано, чтобы показать, как такое вычисление производится для псевдозаголовка UDP. Для простоты все необходимые поля копируются во временный символьный буфер — *buf*, после чего он просто передается вместе с длиной буфера в функцию вычисления контрольной суммы.

Наконец, перед отправкой дейтаграммы следует объединить разные части сообщения в непрерывный буфер. Чтобы скопировать заголовки IP и UDP, а потом и данные в непрерывный буфер, примените функцию *memcpy*, после чего вызовите функцию *sendto*, чтобы отправить данные. Когда вы используете параметр *IP\_HDRINCL*, параметр *to* (кому) в *sendto* игнорируется: данные всегда отправляются узлу, который указан в заголовке IP.

Чтобы увидеть Iphdrinc.c в действии, используйте программу получателя UDP из главы 7 — Receiver.c. Например, введите при ее запуске параметры:

```
Receiver.exe -p:5150 -i:xxx.xxx.xxx.xxx -n:5 -b:1000
```

Можете отбросить параметр *-i*, если на компьютере только один сетевой интерфейс, в ином случае укажите IP-адрес одного из интерфейсов. Если на вашем компьютере установлена Windows 2000, запустите программу Iphdrinc.exe:

```
Iphdrinc.exe -fi: 1.2.3.4 -fp: 10 -tuxxx.xxx.xxx.xxx -tp: 5150 -n: 5150
```

IP-адрес, введенный для параметра *-ti*, должен совпадать с тем, на котором слушает получатель. Тогда программа получателя выдаст сообщение:

```
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
```

По завершении вызова *recvfrom*, Receiver с выводит сообщение об этом вместе с адресной информацией, возвращенной из структуры *SOCKADDRIN*, которая передана в *recvfrom*. Проверить, что пакет передается именно с вашим заголовком, можно с помощью Microsoft Network Monitor. Возможные параметры командной строки для примера Iphdnnc таковы:

- **-fi:xxx.xxx.xxx.xxx** — IP-адрес отправителя пакета;
- **-fp:int** — номер порта отправителя пакета;

**Я** -ti:xxx.xxx.xxx.xxx — IP-адрес пункта назначения пакета,

К **-tp:int** — порт получателя пакета,

**M -n:int** — количество отправляемых UDP-дейтаграмм,

**III -m:string** — сообщение для отправки

Таким образом, вы можете задать исходный и целевой IP-адрес и номер порта

## Резюме

Простые сокеты — мощный механизм для манипулирования базовым протоколом. Мы рассмотрели использование простых сокетов лишь для создания программ ICMP и IGMP средствами Winsock, хотя они могут пригодиться в самых разных приложениях. Оценить все преимущества применения простых сокетов и параметра *IPHDRINCL* вы сможете, лишь тщательно изучив протокол IP и другие вложенные в него протоколы.

MT  
Π£  
i ЭK  
)*ft*

W1

$>I$  iflfibrtql

>III	?i	)J
		•]
	es	M
		r]

# Интерфейс Winsock 2 SPI

Можно сказать, что по сравнению с Winsock 2 API, Winsock 2 Service Provider Interface (SPI) представляет другую сторону программирования для Winsock. С одной стороны — у вас API, с другой — SPI. В главах 6-13 рассматривался интерфейс Winsock 2 API. Winsock 2 предназначен для архитектуры *открытых систем Windows* (Windows Open System Architecture, WOSA), обладающей стандартными API-интерфейсами между Winsock и приложениями Winsock, а также SPI-интерфейсами между Winsock и поставщиками служб Winsock (такими как TCP/IP). На рис. 14-1 показано, как библиотека Ws2\_32.dll из состава Winsock 2 посредничает между приложениями и поставщиками служб Winsock.



**Рис. 14-1. WOSA-архитектура Winsock 2**

В этой главе мы подробно рассмотрим Winsock 2 SPI и то, как разработать поставщик службы, расширив тем самым возможности Winsock 2.

## Основы SPI

Winsock 2 SPI позволяет разработать два типа поставщиков службы поставщик транспорта и поставщик пространства имен *Поставщики транспорта* (transport service provider, TSP), обычно называемые наборами протоколов (типа TCP/IP) — это службы, предоставляющие функции установления связи, передачи данных, управления потоком, контроля ошибок и т.п. *Поставщики пространства имен* (name space provider, NSP) — это службы, которые сопоставляют атрибуты адресации сетевого протокола понятным именам, и таким образом обеспечивают независимое от протокола разрешение имен. В этой роли выступают поддерживаемые Win32 библиотеки DLL, подключаемые к модулю Ws2\_32.dll. Они обеспечивают внутреннее функционирование многих вызовов, определенных в Winsock 2 API. >

### Соглашения SPI об именах

Для прототипов функций Winsock 2 SPI действуют соглашения об именах префиксов. Они определяют функции:

**III WSP (поставщик Winsock)** — поставщика транспортной службы,

- **NSP (поставщик пространства имен)** — поставщика пространства имен,
- **WPU (обратный вызов поставщика Winsock)** — поддержки Ws2\_32.dll, вызываемые поставщиками служб,

**III\* WSC (конфигурация Winsock)** — установки поставщиков служб в Winsock 2

Например, функция с именем *WSCInstallProvider* — это функция конфигурации SPI.

### Соответствие функций Winsock 2 API и SPI

В большинстве случаев, когда приложение вызывает функцию Winsock 2, библиотека Ws2\_32.dll вызывает соответствующую функцию Winsock 2 SPI для выполнения необходимых операций с использованием определенного поставщика службы. Например, *select* соответствует *WSPSelect*, *WSAConnect* — *WSPConnect*, а *WSAAccept* — *WSPAccept*. Однако не для всех функций Winsock есть соответствующая функция SPI.

**III Вспомогательные функции** типа *htonl*, *htons*, *ntohl* и *ntohs* реализованы в рамках Ws2\_32.dll и не передаются поставщику службы. То же верно для WSA-версий этих функций.

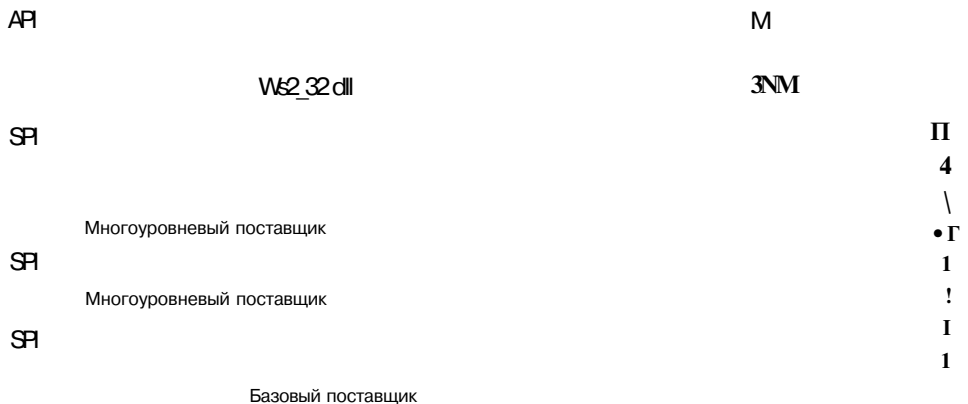
- **Функции преобразования IP** типа *metjxaddr* и *inetjitoa* реализованы только в рамках Ws2\_32.dll
- **Все относящиеся к IP функции преобразования и разрешения имен** в Winsock I I *getxbyy*, *WSAAsyncGetXByY* и *WSACancelAsyncRequest*, а также *gethostname*, — реализованы в рамках Ws2\_32.dll.

- **Winsock-функции поставщика службы, ответственные за пере-числение и блокирующее подключение**, реализованы в библиотеке Ws2\_32.dll. Таким образом, функции *WSAEnumProtocols*, *WSAIsBlocking*, *WSASetBlockingHook* и *WSAUnhookBlockingHook* не требуются в SPI.
- **Коды ошибок Winsock** обрабатываются в рамках Ws2\_32.dll. Функциям *WSAGetLastError* и *WSASetLastError* не требуются в SPI.
- **Функции манипуляции объектами событий и ожидания**, включая *WSACreateEvent*, *WSACloseEvent*, *WSASetEvent*, *WSAResetEvent* и *WSAWaitForMultipleEvents* — напрямую проецируются в вызовы Win32 и не существуют в SPI.

Теперь посмотрим, какие Winsock API соответствуют поставщикам службы Winsock 2. Все определения прототипов функций SPI вы найдете в заголовочном файле Ws2spi.h.

## Поставщики транспортной службы

В Winsock 2 используются поставщики транспортной службы двух типов: базовые и многоуровневые. *Базовые* (base) поставщики службы реализуют конкретные детали сетевого транспортного протокола (типа TCP/IP), включая базовые сетевые функции протокола, такие как отправка и получение данных по сети. *Многоуровневые* (layered) поставщики службы реализуют только высокоуровневые пользовательские функции связи и нуждаются в базовом поставщике службы для фактического обмена данными по сети. Например, вы можете написать код диспетчера безопасности данных или пропускной способности поверх существующего базового поставщика TCP/IP. На рис. 14-2 показано, как установить одноуровневые или многоуровневые поставщики между Ws2\_32.dll и базовым поставщиком.



**Рис. 14-2. Архитектура многоуровневого поставщика**

Этот раздел посвящен основным принципам разработки поставщика транспортной службы. Однако мы не будем излагать все подробности реализации конкретных функций SPI, например, записи SPI-функцией *WSPSend*.



данных в сетевой адаптер. Вместо этого рассмотрим, как функция *WSPSend* многоуровневого поставщика вызывает функцию *WSPSend* нижестоящего поставщика, что является требованием большинства многоуровневых поставщиков службы. По существу, разработка многоуровневого поставщика во многом сводится к передаче SPI-вызовов вашего поставщика следующему нижестоящему поставщику.

Мы также обсудим сложности в обработке вызовов ввода-вывода, в рамках описанных в главе 8 моделей Winsock. На прилагаемом компакт-диске приведен пример под названием LSP, который демонстрирует, как реализовать многоуровневый поставщик службы, просто рассчитывающий, сколько байтов передано через сокет с использованием транспортного протокола IP. Microsoft Platform SDK предоставляет более сложный пример многоуровневого поставщика службы, называемого *layered*. Его можно найти в примерах MSDN Platform SDK по адресу <ftp://ftp.microsoft.com/bussys/WinSock/winsock2/layeredzip>.

**ПРИМЕЧАНИЕ** В этом разделе, посвященном поставщикам транспортной службы, мы часто используем термины «клиент SPI» и «нижестоящий поставщик». Клиентом SPI может быть библиотека Winsock 2 *Ws2\_32.dll* или другой многоуровневый поставщик службы, вышестоящий по отношению к вашему поставщику. Клиент SPI никогда не является самим приложением Winsock, так как приложения Winsock должны использовать Winsock 2 API, экспортируемый из *Ws2\_32.dll*.

Термин «нижестоящий поставщик» используется только при описании аспектов разработки многоуровневого поставщика службы. Нижестоящий поставщик может быть другим многоуровневым или основным поставщиком службы. На компьютере порой устанавливают множество многоуровневых поставщиков службы, так что многоуровневый поставщик может быть ниже вашего. !ЯЛ

## Функция *WSPStartup*

Поставщики транспортной службы Winsock 2 реализованы как стандартные модули динамически компокуемой библиотеки Windows, в которые вы должны экспортировать функцию *DUMain*. Дополнительно следует экспортировать одну запись функции с именем *WSPStartup*. Когда SPI-клиент вызывает *WSPStartup*, он предоставляет 30 дополнительных SPI-функций, которые составляют поставщик транспортной службы через переданную в качестве параметра диспетчерскую таблицу функций (табл. 14-1). Ваш поставщик службы должен обеспечить реализацию *WSPStartup* и всех 30 функций.

Как и когда вызывается функция *WSPStartup*? Может показаться, что когда приложение вызывает *WSAStartup* API. Но это не так, Winsock не знает тип поставщика службы, который нужно использовать в ходе выполнения *WSAStartup*, а определяет, какой поставщик службы нужно загрузить, основываясь на семействе адреса, типе сокета и параметрах протокола из вызова *WSASocket*. Поэтому Winsock вызывает поставщик службы, только когда приложение создает сокет через вызов функций *socket* или *WSASocket*. Например,

если приложение создает сокет с использованием семейства адресов *AFJNET* и типа сокета *SOCK\_STREAM*, Winsock ищет и загружает соответствующий поставщик транспорта, который обеспечивает функциональность TCP/IP. Подробнее процесс загрузки будет описан далее в этой главе, в разделе, посвященном установке поставщиков транспортной службы.

**Табл. 14-1. Функции поддержки поставщика транспорта**

Функция API	Соответствующая функция SPI
<i>WSAAccept</i> ( <i>accept</i> также косвенно соответствует <i>WSPAccept</i> )	<i>WSPAccept</i>
<i>WSAAddressToString</i>	<i>WSPAddressToString</i>
<i>WSAAsyncSelect</i>	<i>WSPAsyncSelect</i>
<i>bind</i>	<i>WSPBind</i>
<i>WSACancelBlockingCall</i>	<i>WSPCancelBlockingCall</i>
<i>WSACleanup</i>	<i>WSPCleanup</i>
<i>dosesocket</i>	<i>WSPCloseSocket</i>
<i>WSAConnect</i> ( <i>connect</i> также косвенно соответствует <i>WSPConnect</i> )	<i>WSPConnect</i>
<i>WSADuplicateSocket</i>	<i>WSPDuplicateSocket</i>
<i>WSAEnumNetworkEvents</i>	<i>WSPEnumNetworkEvents</i>
<i>WSAEventSelect</i>	<i>WSPEventSelect</i>
<i>WSAGetOverlappedResult</i>	<i>WSPGetOverlappedResult</i>
<i>getpeername</i>	<i>WSPGetPeerName</i>
<i>getsockname</i>	<i>WSPGetSockName</i>
<i>getsockopt</i>	<i>WSPGetSockOpt</i>
<i>WSAGetQOSByName</i>	<i>WSPGetQOSByName</i>
<i>WSAIoctl</i>	<i>WSPIoctl</i>
<i>WSAJoinLeaf</i>	<i>WSPJoinLeaf</i>
<i>listen</i>	<i>WSPListen</i>
<i>WSARecv</i> ( <i>recv</i> также косвенно соответствует <i>WSPRecv</i> )	<i>WSPRecv</i>
<i>WSARecvDisconnect</i>	<i>WSPRecvDisconnect</i>
<i>WSARecvFrom</i> ( <i>recvfrom</i> также косвенно соответствует <i>WSPRecvFrom</i> )	<i>WSPRecvFrom</i>
<i>select</i>	<i>WSPSelect</i>
<i>WSASend</i> ( <i>send</i> также косвенно соответствует <i>WSPSend</i> )	<i>WSPSend</i>
<i>WSASendDisconnect</i>	<i>WSPSendDisconnect</i>
<i>WSASendTo</i> ( <i>sendto</i> также косвенно соответствует <i>WSPSendTo</i> )	<i>WSPSendTo</i>
<i>setsockopt</i>	<i>WSPSetSockOpt</i>
<i>shutdown</i>	<i>WSPShutdown</i>
<i>WSASocket</i> ( <i>socket</i> также косвенно соответствует <i>WSPSocket</i> )	<i>WSPSocket</i>
<i>WSAStringToAddress</i>	<i>WSPStringToAddress</i>

## Параметры *WSPStartup*

*WSPScirtup* — ключевая функция, используемая для инициализации поставщика транспортной службы. Она определена так.

```
int WSPStartup(
    WORD wVersionRequested,
    LPWSPDATA lpWSPData,
    LPWSA_PROTOCOL_INFO lpProtocolInfo,
    WSUPCALLTABLE UpoallTable,
    LPWSFFOC_TABLE lpProcTable
);
```

Параметр *wVersionRequested* получает номер последней версии Windows Sockets SPI, которую может использовать вызывающая программа. Ваш поставщик службы должен проверить это значение, чтобы убедиться, что нужная версия поддерживается. Он использует параметр *lpWSPData* для возвращения информации о своей версии через структуру *WSPDATA*.

```
typedef struct WSPData
{
    WORD        wVersion;
    WORD        wHighVersion;
    WCHAR        szDescription[WSPDESCRIPTION_LEN + 1];
} WSPDATA, FAR * LPWSPDATA;
```

В поле *wVersion* поставщик должен вернуть версию Winsock, которую, как ожидается, будет использовать вызывающая программа. Параметр *wHighVersion* возвращает самую последнюю версию Winsock, поддерживаемую поставщиком. Это обычно то же значение, что и в поле *wVersion* (информация по версиям Winsock подробно рассмотрена в главе 7). Поле *szDescription* возвращает строку UNICODE, завершающуюся символом null и определяющую ваш поставщик для SPI-клиента. Это поле может содержать до 256 символов.

Параметр *lpProtocolInfo* функции *WSPStartup* — указатель на структуру *WSAPROTOCOL\_INFOW*, которая содержит информацию о поставщике (характеристики протокола и детали этой структуры описаны в главе 5). Информация в *WSAPROTOCOL\_INFOW* извлекается библиотекой *Ws2\_32.dll* из каталога поставщика служб Winsock 2, содержащего сведения о свойствах поставщиков служб. Мы рассмотрим записи каталога Winsock 2 в разделе, посвященном установке поставщиков транспортных служб.

Разрабатывая многоуровневый поставщик служб, вы должны обрабатывать параметр *lpProtocolInfo* уникально, ведь он задает способ, которым ваш поставщик посредничает между *Ws2\_32.dll* и основным поставщиком служб. Параметр используется для определения следующего поставщика служб, нижестоящего по отношению к вашему (это может быть другой многоуровневый или базовый поставщик). В какой-то момент ваш поставщик должен загрузить следующий поставщик служб, загрузив его модуль DLL и вызвав функцию поставщика *WSPStartup*. Структура *WSAPROTOCOL\_INFOW*, на которую указывает *lpProtocolInfo*, содержит поле *ProtocolChain*, определяющее место вашего поставщика служб в ряду других.

Поле *ProtocolChain* — фактически структура *WSAPROTOCOLCHAIN*, определенная как

```
typedef struct WSAPROTOCOLCHAIN
<
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

Поле *ChainLen* определяет, сколько уровней между *Ws2\_32.dll* и основным поставщиком службы (включительно). Если на компьютере только один многоуровневый поставщик службы поверх протокола (такого как TCP/IP), это значение равно 2. Поле *ChainEntries* — массив идентифицирующих номеров каталога поставщика службы, уникально определяющих многоуровневых поставщиков службы, связанных вместе для конкретного протокола. Мы опишем структуру *WSAPROTOCOLCHAIN* далее в этой главе, в разделе по установке поставщиков транспортной службы.

Многоуровневый поставщик службы требует, в частности, искать поле *ProtocolChain* этой структуры, чтобы выяснить ее расположение в массиве поставщиков службы (с помощью поиска записи каталога вашего уровня), а также определить следующий поставщик в массиве. Если следующий поставщик — другого уровня, передайте структуру *ipProtocolInfo* функции *WSPStartup* следующего уровня без изменений. Если следующий уровень — последний элемент в массиве (то есть базовый поставщик), ваш поставщик должен использовать структуру *WSAPROTOCOLINFO* базового поставщика для подстановки в структуру *IpProtocolInfo* при вызове функции основного поставщика *WSPStartup*. Листинг 14-1 демонстрирует, как многоуровневый поставщик программно управляет структурой *IpProtocolInfo*.

#### Листинг 14-1. Поиск соответствующей структуры *WSAPROTOCOLINFO* для функции *WSPStartup*

```
LPWSAPROTOCOLINFO ProtocollInfo;
LPWSAPROTOCOLINFO ProtolInfo = IpProtocolInfo;
DWORD ProtocollInfoSize = 0;

// Определение, как много записей нам нужно перечислить
if (WSCEnumProtocols(NULL, ProtocollInfo, &ProtocollInfoSize,
    &ErrorCode) == SOCKET.ERROR)
{
    if (ErrorCode != WSAENOBUFFS)
    {
        return WSAEPROVIDERFAILEDINIT;
    }

    if ((ProtocollInfo = (LPWSAPROTOCOLINFO) GlobalAlloc(GPTR,
        ProtocollInfoSize)) == NULL)
    {
        return WSAEPROVIDERFAILEDINIT;
```

'si',  
itk

см. след. стр.

**Листинг 14-1.**    (продолжение)

&gt;; •&lt;

```

if ((TotalProtocols = WSCEnumProtocols(NULL, ProtocolInfo,
    &ProtocolInfoSize, &ErrorCode)) == SOCKET_ERROR)
{
    return WSAEPROVIDERFAILEDINIT;

// Поиск идентификационной записи нашего многоуровневого поставщика в каталоге      i
for (l=0; l < TotalProtocols; l++)
{
    if (memcmp (&ProtocolInfo[l].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
    {
        gLayerCatId = ProtocolInfo[l].dwCatalogEntryId;
        break;

// Сохранение идентификационной записи нашего поставщика в каталоге
gChainId = lpProtocolInfo->dwCatalogEntryId;

// Поиск нашей идентификационной записи в цепочке протоколов
for(j = 0; j < lpProtocolInfo->ProtocolChain.ChainLen; j++)
{
    if (lpProtocolInfo->ProtocolChain.ChainEntries[j] ==
        gLayerCatId)
    {
        NextProviderCatId =
            lpProtocolInfo->ProtocolChain.ChainEntries[j+1];

        // Проверка, является ли следующий поставщик базовый
        if (lpProtocolInfo->ProtocolChain.ChainLen ==
            0 + 2) )
        {
            for (l=0, l < TotalProtocols; l++)
                if (NextProviderCatId ==
                    ProtocolInfo[l].dwCatalogEntryId)
                {
                    ProtocolInfo = &ProtocolInfo[l];
                    break;

                break;

// В этот момент ProtocolInfo будет содержать соответствующую
// структуру WSAPROTOCOL_INFO

```

Параметр *UpcallTable* функции *WSPStartup* получает диспетчерскую таблицу обратного вызова библиотеки SPI Ws2\_32.dll, содержащую указатели на вспомогательные функции, которые ваш поставщик может использовать для управления операциями ввода-вывода между ним и Winsock 2. Мы определим большинство этих функций и опишем, как их использовать, далее в этой главе, в разделе, посвященном поддержке модели ввода-вывода Winsock.

Заключительный параметр *WSPStartup* — *ipProcTable*, представляет собой таблицу из 30 указателей на SPI-функции, которые должен реализовать ваш поставщик службы (табл. 14-1). Каждая SPI-функция соответствует спецификации параметров его API-аналога, за исключением следующих изменений. Во-первых, в каждой функции есть заключительный параметр *ipErrno*, который поставщик должен использовать, чтобы сообщить код ошибки Winsock, если реализация неудачна. Например, если вы реализуете *WSPSend* и не можете выделить память, пусть вернется ошибка *WSAENOBUFS*.

Кроме того, функции SPI *WSPSend*, *WSPSendTo*, *WSPRecv*, *WSPRecvFrom* и *WSPIOCTL* имеют дополнительный параметр *ipThreadld*, который определяет поток приложения, вызывающий функцию SPI. Эта особенность полезна для поддержки процедур завершения.

И последнее несколько функций Winsock 1.1, такие как *send* и *recv*, прямо соответствуют функциям Winsock 2. Эти функции Winsock 1.1 не соответствуют прямо SPI-функциям, так как фактически вызывают функцию Winsock 2, обладающую подобными возможностями. Например, функция *send* фактически вызывает функцию *WSASend*, которая соответствует *WSPSend* (табл. 14-1).

### Счетчик экземпляров

В спецификации Winsock приложения могут вызывать функции *WSAStartup* и *WSACleanup* неограниченное число раз. Функции *WSAStartup* и *WSACleanup* вашего поставщика службы будут вызываться столько же раз, сколько их API-аналоги. В результате, ваш поставщик службы должен поддерживать счетчик экземпляров, указывающий, сколько раз вызвана *WSPStartup*. Вам следует убавлять это значение на 1 для каждого вызова *WSPCleanup*. Цель поддержки счетчика экземпляров — упростить инициализацию и очистку поставщика службы. Например, если ваш поставщик выделяет память для управления внутренними структурами, вы можете удерживать эту память, пока счетчик экземпляров больше 0. Когда он опустится до 0, Ws2\_32.dll выгрузит ваш поставщик из памяти.

### Описатели сокетов

Поставщик службы должен возвращать описатели сокетов при вызове SPI-клиентом функций *WSPSocket*, *WSPAccept* и *WSPJoinLeaf*. Это могут быть IFS-описатели, тогда поставщик службы называется IFS-поставщиком. Все базовые поставщики транспорта Microsoft — IFS-поставщики.

Winsock разработан так, чтобы позволить приложениям Winsock использовать функции Win32 API *ReadFile* и *WriteFile* для получения и отправки данных по описателю сокета. Поэтому вы должны учитывать, как в поставщике службы создаются описатели сокета. Для приложений, вызывающих *ReadFile*

и *WriteFile* на описателе сокета, лучше разработать IFS-поставщик, но при этом не забудьте об ограничениях ввода-вывода.

### IFS-поставщик

Как мы уже упоминали, поставщики транспорта могут быть многоуровневыми или базовыми. Если вы разрабатываете базовый IFS-поставщик, он будет иметь компонент ОС режима ядра, и это позволит поставщику Winsock создавать описатели, которые могут использоваться подобно описателям файловой системы в вызовах *ReadFile* и *WriteFile*. Разработка ПО режима ядра — за рамками этой книги. Если вы хотите узнать о ней больше, см. MSDN Device Development Kit (DDK).

Многоуровневый поставщик службы также может стать IFS-поставщиком, но только если расположен поверх существующего базового IFS-поставщика. При этом описатель сокета нижестоящего IFS-поставщика, полученный в вашем многоуровневом поставщике, напрямую передается SPI-клиенту. Передача описателей сокета напрямую от нижестоящего поставщика ограничивает функциональные возможности многоуровневого поставщика следующим образом.

Прежде всего, функции *WSPSend* и *WSRecv* многоуровневого поставщика не будут вызываться, если функции *ReadFile* и *WriteFile* вызываются по сокету. Эти функции обойдут многоуровневого поставщика и напрямую вызовут реализацию базового IFS-поставщика.

Многоуровневый поставщик также не сможет выполнить заключительную обработку запросов перекрытого ввода-вывода, поступивших на порт завершения. Заключительная обработка порта завершения полностью обходит многоуровневого поставщика.

Если ваш многоуровневый поставщик должен контролировать весь ввод-вывод, придется разработать многоуровневый Не-IFS-поставщик.

Всякий раз, когда IFS-поставщик (многоуровневый или базовый) создает новый описатель сокета, требуется поставщик для вызова *WPUModifyIFSHandle* до предоставления нового описателя SPI-клиенту. Это позволяет Winsock Ws2\_32.dll существенно упростить процесс идентификации связанного с данным сокетом IFS-поставщика службы, когда Win32 API (такие как *ReadFile* и *WriteFile*) выполняют ввод-вывод через сокет. *WPUModifyIFSHandle* определена таю

```
SOCKET WPUModifyIFSHandle(
    DWORD dwCatalogEntryId,           ^           ,,Q
    SOCKET ProposedHandle,
    LPINT lpErrno                      -, f,
```

Поле *dwCatalogEntryId* определяет идентификатор каталога вашего поставщика службы. Параметр *ProposedHandle* представляет IFS-описатель, выделенный этим поставщиком (если он базовый). Если вы разрабатываете многоуровневый IFS-поставщик, этот описатель будет передаваться от нижестоящего поставщика. Параметр *lpErrno* получает информацию о конкретном коде ошибки Winsock, если эти функции дают сбой со значением *INVALID\_SOCKET*.

## Не-IFS-поставщик

Если вы разрабатываете многоуровневый поставщик и намерены контролировать каждую проходящую через сокет операцию чтения и записи, вам лучше выбрать Не-IFS-поставщик. Не-IFS-поставщики создают описатели сокета, используя обратный вызов *WPUCreateSocketHandle*. Описатели сокета, созданные *WPUCreateSocketHandle*, подобны описателям IFS-поставщика, так как позволяют приложениям Winsock использовать функции *ReadFile* и *WriteFile* на сокете. Однако с этими двумя функциями связана существенная проблема ввода-вывода, так как архитектура Winsock 2 должна выполнять переадресацию ввода-вывода функциям *WSPRecv* и *WSPSend* поставщика службы. *WPUCreateSocketHandle* определена таю

```
SOCKET WPUCreateSocketHandle(
    DWORD dwCatalogEntryId,
    DWORD dwContext,
    LPINT lpErrno
```

Поле *dwCatalogEntryId* определяет идентификатор каталога вашего поставщика службы. Параметр *dwContext* позволяет сопоставлять данные поставщика с описателем сокета. Заметьте: *dwContext* дает большую свободу в отношении информации, которую вы можете связать с описателем сокета. В примере LSP на прилагаемом компакт-диске это поле используется для сохранения количества переданных и полученных данных. Winsock обеспечивает обратный вызов функции *WPUQuerySocketHandleContext*, которая может применяться для извлечения связанных с сокетом данных поставщика, сохраненных в поле *dwContext*. Этот обратный вызов определен так:

```
int WPUQuerySocketHandleContext(
    SOCKET s,
    LPDWORD lpContext,
    LPINT lpErrno
```

Параметр 5 — описатель сокета, переданный SPI-клиентом (первоначально созданный *WPUCreateSocketHandle*), для которого вы хотите восстановить сокетные данные поставщика. Параметр *lpContext* получает данные поставщика, первоначально переданные *WPUCreateSocketHandle*. Параметр *lpErrno* в обеих функциях получает определенный код ошибки Winsock, если выполнение этих функций не удастся. Если выполняется *WPUCreateSocketHandle*, возвращается значение *INVALID\_SOCKET*, если *WPUQuerySocketHandleContext* — значение *SOCKETERROR*.

## Поддержка модели ввода-вывода Winsock

Winsock предлагает несколько моделей, которые могут использоваться для управления вводом-выводом через сокет. С точки зрения поставщика служб, каждая модель требует обратного вызова какой-либо SPI-функции из библиотеки *Ws2\_32.dll*, доступной из уже упомянутого параметра *UpcallTable* в



*WSPStartup*. Если вы разрабатываете простой многоуровневый IFS-поставщик, принципы, описанные в следующих разделах, не применяются — вы просто полагаетесь на нижестоящий поставщик, чтобы управлять всем вводом-выводом для каждой модели. Рассматриваемые принципы относятся к любому другому типу поставщика. Мы сосредоточимся на аспектах разработки многоуровневого Не-IFS-поставщика службы.

### Блокирующий и неблокирующий ввод-вывод

Блокирующий ввод-вывод — самая простая форма ввода-вывода в Winsock 2. Любая операция ввода-вывода с блокируемым сокетом не будет возвращена, пока не завершится. Поэтому любой поток может выполнять одновременно только одну операцию ввода-вывода. Например, когда SPI-клиент вызывает функцию *WSPRecv* в блокируемом режиме, ваш поставщик должен только передать запрос прямо вызову *WSPRecv* следующего поставщика. Функция *WSPRecv* вашего поставщика вернется, только когда вызов *WSPRecv* завершит следующий поставщик.

Реализация блокируемого ввода-вывода не сложна, но все же обратите внимание на обратную совместимость с блокирующими вызовами Winsock 1.1. Вызовы *WSASetBlockingCall* и *WSACancelBlocking* API удалены из спецификации Winsock 2. Впрочем, *WSPCancelBlockingHook* все-таки может быть вызвана библиотекой *Ws2\_32.dll*, если приложение Winsock 1.1 вызывает функции *WSASetBlockingHook* и *WSACancelBlockingCall*. В многоуровневом поставщике службы вы можете просто передать вызов *WSPCancelBlockingHook* запросу основного поставщика. Если вы реализуете основной поставщик, и происходит запрос блокирования, следует предусмотреть механизм периодического вызова функции *WPUQueryBlockingCallback*:

```
int WPUQueryBlockingCallback(
    DWORD dwCatalogEntryId,
    LPBLOCKINGCALLBACK FAR lpIpfnCallback,
    LPDWORD lpdwContext,
    LPKT lpErrno
);
```

Параметр *dwCatalogEntryId* получает идентификатор записи каталога вашего поставщика, как описано в функции *WSPStartup*. Параметр *lpIpfnCallback* — указатель на функцию обработчика блокирующих прерываний приложения, которую вы должны периодически вызывать для предотвращения реального блокирования блокирующих вызовов приложения. Эта функция обратного вызова имеет следующую форму:

```
typedef BOOL (CALLBACK FAR * LPBLOCKINGCALLBACK)(
    DWORD dwContext
);
```

Когда ваш поставщик периодически вызывает *IpIpfnCallback*, значение *lpdwContext* передается в параметр функции обратного вызова *dwContext*. Заключительный параметр *WPUQueryBlockingCallback* — *lpErrno*, возвращает код ошибки Winsock, если функции возвращают значение *SOCKETJERROR*.

## Модель *select*

Модель ввода-вывода *select* требует, чтобы поставщик управлял структурами *fdset* для параметров *readfds*, *writefds* и *exceptfds* функции *WSPSelect*:

```
int WSPSelect(
    int nfd,
    fd_set FAR *readfds,
    fd_set FAR *writefds,
    fd_set FAR *exceptfds,
    const struct timeval FAR *timeout,
    LPINT lpErrno
);
```

По сути, тип данных *fd\_set* представляет собой совокупность сокетов. Когда клиент вызывает ваш поставщик, используя *WSPSelect*, он передает описатели сокета одному или нескольким этим наборам. Поставщик отвечает за определение сетевой активности на каждом из перечисленных сокетов.

Для многоуровневых Не-IFS-поставщиков это требует создания трех полей данных *fdset* и привязки описателей сокета клиента SPI к описателям сокета нижестоящего поставщика в каждом наборе. Как только все наборы определены, поставщик вызывает функцию *WSPSelect* нижестоящего поставщика. Когда нижестоящий поставщик завершает работу, ваш поставщик должен определить, на каких сокетах есть события, ожидающие очереди в каждом из полей *fdset*. Существует полезный обратный вызов *WPUFDIsSet*, который определяет установленные сокетные нижестоящего поставщика. Этот обратный вызов подобен макрокоманде *FDJSET* (см. главу 8):

```
int WPUFDIsSet(
    SOCKET s,
    FD_SET FAR *set
);
```

Параметр *s* — сокет, который вы ищете в наборе. Параметр *set* — фактический набор описателей сокета. Проверив содержимое каждого набора, переданное нижестоящему поставщику, ваш поставщик должен сохранить соответствие сокета верхнего поставщика нижестоящему. Когда нижестоящий поставщик завершает вызов *WSPSelect*, легко определить, какие сокеты ожидают ввода-вывода для вышестоящего поставщика.

Определив сокеты нижестоящего поставщика, у которых в очереди есть сетевые события, обновите исходные наборы *fdset*, переданные исходным SPI-клиентом. Файл *Ws2spi.h* описывает три макрокоманды — *FD\_CLR*, *FD\_SET* и *FD\_ZERO*, которые могут использоваться для управления исходными наборами (см. главу 8). Листинг 14-2 демонстрирует одну из возможных реализаций *WSPSelect*.

### Листинг 14-2. Подробности реализации *WSPSelect*

```
int WSPAPI WSPSelect(
    int nfd,
```

см. след. стр.

## Листинг 14-2. {продолжение)

```

    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout,
    LPINT lpErrno)
{
    SOCK_INFO *SocketContext;
    u_int I;
    u_int count;
    int Ret;
    int HandleCount;

    // Построение таблицы привязок сокетов для вышестоящего и нижестоящего
    поставщиков
    struct
    {
        SOCKET ClientSocket;
        SOCKET ProvSocket;
    > Read[FD_SETSIZE], Wnte[FD_SETSIZE], Except[FD_SETSIZE];

    fd_set ReadFds, WriteFds, ExceptFds;

    // Построение набора ReadFds для нижестоящего поставщика
    if (readfds)
    {
        FD_ZERO(&ReadFds);

        for (i=0; i < readfds->fd_count; i++)

            if (MainUpCallTable.lpWPUQuerySocketHandleContext(
                (Read[i].ClientSocket = readfds->fd_array[i]),
                (LPDWORD) &SocketContext, lpErrno) ==
                SOCKET_ERROR)
                return SOCKET_ERROR;
            FD_SET(Read[i].ProvSocket =
                SocketContext->ProviderSocket), &ReadFds);

    // Построение набора WriteFds для нижестоящего поставщика.
    // Это в точности подобно построению набора ReadFds выше.

    // Построение набора ExceptFds для нижестоящего поставщика.
    // Это также подобно построению набора ReadFds выше.

    // Вызов функции WSPSelect нижестоящего поставщика
    Ret = NextProcTable.lpWSPSelect(nfds,

```

Листинг 14-2. (продолжение)

```
(readfds ? &ReadFds  NULL),
(writefds ? iWnteFds : NULL),
(exceptfds ' &ExceptFds : NULL), timeout, lpErrno);

if (Ret != SOCKET_ERROR)
{
    HandleCount = Ret;

    // Настройка набора readfds вызывающего поставщика
    if (readfds)
    {
        count = readfds->fd_count;
        FD_ZERO(readfds);

        for(i =0; (i < count) && HandleCount; i++)
        <
            if (MamUpCallTable.lpWPUFDIsSet(
                Read[i].ProvSocket, &ReadFds))

                FD_SET(Read[i].ClientSocket, readfds);
                HandleCount--;

        // Настройка набора writefds вызывающего поставщика.
        // Это в точности подобно настройке набора readfds выше.

        // Настройка набора exceptfds вызывающего поставщика.
        // Это также подобно настройке набора readfds выше.

    }

    return Ret;
}
```

## Модель WSAAsyncSelect

Эта модель ввода-вывода описывает основанное на сообщениях Windows уведомление о сетевых событиях на соquete. SPI-клиенты используют ее, вызывая функцию *WSPAsyncSelect*:

```
int WSPAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent,
    LPINT lpErrno
```

Параметр *s* представляет сокет SPI-клиента, ожидающий сообщения Windows о сетевых событиях. Параметр *hWnd* задает дескриптор окна, которое должно получить сообщение, определенное параметром *wMsg*, когда на сокете произойдет сетевое событие (определенное параметром *lEvent*). Параметр *ipErrno* получает код ошибки Winsock, если реализация этой функции возвращает *SOCKET\_ERROR*. Сетевые события, которые ваш поставщик должен поддерживать, заданы параметром *lEvent* (см. главу 8).

Когда клиент вызывает *WSPAsyncSelect*, ваш поставщик отвечает за уведомление клиента о происходящих на сокете сетевых событиях, используя предоставленные клиентом дескриптор окна и сообщение (переданы через вызов *WSPAsyncSelect*). Ваш поставщик может уведомлять клиента о сетевых событиях, вызывая функцию *WPUPostMessage*:

```
BOOL WPUPostMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

Поставщик использует параметр *hWnd*, чтобы пометить как занятый дескриптор окна клиента SPI. Параметр *Msg* определяет заданное пользователем сообщение, которое первоначально передано в параметр *wMsg* функции *WSPAsyncSelect*. Параметр *WParam* принимает дескриптор сокета, на котором произошло сетевое событие. Последний параметр — *lParam*, состоит из двух частей: нижняя содержит информацию о произошедшем сетевом событии, верхняя — код ошибки Winsock (если произошла ошибка, связанная с сетевым событием, указанным в нижней части).

Многоуровневый He-IFS-поставщик службы — клиент функции *WSPAsyncSelect* нижестоящего поставщика. Значит, ваш поставщик должен транслировать дескрипторы сокета SPI-клиента в свои дескрипторы сокета с использованием *WPUQuerySocketHandleContext* перед вызовом функции *WSPAsyncSelect* нижестоящего поставщика. Чтобы сообщить в окне SPI-клиента о сетевых событиях, поставщик должен использовать службы *WPUPostMessage*, и следовательно, перехватывать сообщения нижестоящего поставщика. Дело в том, что функция *WPUPostMessage* возвращает дескриптор сокета нижестоящего поставщика в верхнем слове параметра *lParam*, и ваш поставщик должен транслировать дескриптор сокета в верхнем слове *lParam* в дескриптор сокета SPI-клиента, который использовался в исходном вызове *WSPAsyncSelect*.

Лучше всего управлять перехватом оконных сообщений нижестоящего поставщика с помощью рабочего потока, который создает скрытое окно для управления оконными сообщениями о сетевых событиях. Когда ваш поставщик вызывает функцию *WSPAsyncSelect* нижестоящего поставщика, вы просто передаете дескриптор рабочего окна вызову функции *WSPAsyncSelect* нижестоящего поставщика. Теперь когда рабочее окно вашего поставщика получит сообщения о сетевом событии от нижестоящего поставщика, ваш поставщик сообщит о них SPI-клиенту, используя *WPUPostMessage*.

### Модель ввода-вывода *WSAEventSelect*

*WSAEventSelect* включает сигнальные объекты событий, когда на сокете происходят сетевые события. SPI-клиенты используют эту модель, вызывая функцию *WSPEventSelect*:

```
int WSPEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long INetworkEvents,
    LPINT lpErrno
);
```

Параметр *s* — сокет SPI-клиента, который ожидает уведомления о сетевых событиях. Параметр *hEventObject* — описатель объекта *WSAEVENT*, о котором ваш поставщик сообщает, когда на сокете *s* происходят сетевые события, указанные в параметре *INetworkEvents*. Параметр *lpErrno* получает код ошибки Winsock, если эта функция возвращает *SOCKET\_ERROR*. Сетевые события, которые ваш поставщик должен поддерживать (определенные параметром *INetworkEvents*) — те же, что и в модели ввода-вывода *WSAAsyncSelect*.

В многоуровневом Не-IFS-поставщике *WSPEventSelect* реализуется тривиально. Когда SPI-клиент передает объект события, поставщик должен только транслировать описатель сокета SPI-клиента в описатель сокета нижестоящего поставщика, используя функцию *WPUQuerySocketHandleContext*. После трансляции вызовите функцию *WSPEventSelect* нижестоящего поставщика, используя объект события, взятый непосредственно от SPI-клиента. Когда операции ввода-вывода происходят на объекте события SPI-клиента, нижестоящий поставщик прямо сообщает об объекте события, и SPI-клиент уведомляется о сетевых событиях.

Когда нижестоящий поставщик сообщает об объекте события, никакой описатель сокета не возвращается SPI-клиенту после завершения этого события. Поэтому ваш поставщик не должен транслировать описатель сокета для SPI-клиента. Этим модель ввода-вывода *WSAEventSelect* отличается от описанной ранее модели *WSAAsyncSelect*.

При разработке базового поставщика, учтите, что он должен уведомлять SPI-клиента, когда на сокете происходят определенные параметром *INetworkEvents* сетевые события. При этом ваш поставщик использует предоставленный клиентом объект события, переданный в вызове *WSPEventSelect*. Ваш поставщик может уведомлять SPI-клиента о сетевых событиях через перекрестный вызов функции *WPUSetEvent*:

```
BOOL WPUSetEvent (
    WSAEVENT hEvent,
    LPINT lpErrno
);
```

Параметр *hEvent* представляет переданный функцией *WSPEventSelect* описатель объекта события, о котором ваш поставщик должен сообщить. Параметр *lpErrno* возвращает код ошибки Winsock, если функция возвращает *FALSE*.

## Перекрытый ввод-вывод

Модель перекрытого ввода-вывода требует, чтобы ваш поставщик реализовал диспетчер перекрытого ввода-вывода. Этот диспетчер будет обслуживать и события, основанные на объектах, и завершение основанных на процедурах запросов перекрытого ввода-вывода. Перекрытый ввод-вывод Win32 в Winsock используют SPI-функции *WSPSend*, *WSPSendTo*, *WSPRecv*, *WSPRecvFrom*, *WSPIoctl*.

У каждой из этих функций есть три параметра: необязательный указатель на структуру *WSAOVERLAPPED*, необязательный указатель на функцию *WSAOVERLAPPED\_COMPLETION\_ROUTINE* и указатель на структуру *WSATHREADID* — она определяет выполняющий вызов поток приложения.

Структура *WSAOVERLAPPED* — ключевая для связи поставщика службы с SPI-клиентом в ходе операций перекрытого ввода-вывода. Она определена так:

```
typedef struct .WSAOVERLAPPED {
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

Диспетчер перекрытого ввода-вывода отвечает за управление полем *Internal* структуры *WSAOVERLAPPED* для SPI-клиента. В начале перекрытой обработки ваш поставщик службы должен присвоить полю *Internal* значение *WSS\_OPERATION\_IN\_PROGRESS*. Это важно, ведь если SPI-клиент вызывает функцию *WSPGetOverlappedResult*, в то время как ваш поставщик обслуживает ожидающую выполнения перекрытую операцию, значение *WSSOPERATION\_IN\_PROGRESS* может использоваться в *WSPGetOverlappedResult* для определения, идет ли все еще перекрытая операция.

Когда операция ввода-вывода закончена, ваш поставщик присваивает значения полям *OffsetHigh* и *Offset*. Полю *OffsetHigh* присваивается значение кода ошибки Winsock, полученной в результате операции, а полю *Offset* — значение флагов, полученных в результате операций ввода-вывода *WSPRecv* и *WSPRecvFrom*. После настройки этих полей ваш поставщик уведомит клиента SPI о завершении перекрытого запроса через объект события, либо через процедуру завершения (в зависимости от того, как функции ввода-вывода использовали необязательную структуру *WSAOVERLAPPED*).

**События.** При основанном на событиях перекрытом вводе-выводе SPI-клиент вызывает одну из функций ввода-вывода со структурой *WSAOVERLAPPED*, содержащей объект события. Структуре *WSAOVERLAPPED JOOMPLETION\_ROUTINE* должно быть присвоено значение *NULL*. Ваш поставщик службы отвечает за управление запросом перекрытого ввода-вывода. Когда выполнение запроса завершается, поставщик уведомит поток вызывающей программы о завершении запроса ввода-вывода, используя перекрестный вызов функции *WPUCompleteOverlappedRequest*:

```
WSAEVENT WPUCompleteOverlappedRequest(
    SOCKET s,
```

```

LPWSAOVERLAPPED lpOverlapped,
DWORD dwError,
DWORD cbTransferred,
LPINT lpErrno

```

Параметры *s* и *lpOverlapped* представляют исходный сокет и структуру *WSAOVERLAPPED*, которые были первоначально переданы клиентом. Ваш поставщик должен присвоить параметру *dwError* значение, характеризующее состояние завершения запроса перекрытого ввода-вывода, а параметру *cbTransferred* — число переданных в ходе перекрытой операции байт. Последний параметр — *lpErrno*, сообщает код ошибки Winsock, если обращение к этой функции вернуло значение *SOCKETERROR*. Когда выполнение *WPU-CompleteOverlappedRequest* завершается, она присваивает значения двум полям в структуре *WSAOVERLAPPED* SPI-клиента: полю *InternalHigh* — количество переданных и принятых байт *cbTransferred*, а полю *Internal* — значение, отличное от *WSS\_OPERATION\_IN\_PROGRESS*.

В основном на событиях перекрытом вводе-выводе для восстановления результата завершенного перекрытого запроса SPI-клиент в конечном счете вызывает функцию *WSPGetOverlappedResult*:

```

BOOL WSPGetOverlappedResult (                                     UK-
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno
    } if

```

При вызове этой функции ваш поставщик службы должен сообщить о текущем состоянии исходного перекрытого запроса. Как мы уже упоминали, ваш поставщик отвечает за управление полями *Internal*, *InternalHigh*, *Offset* и *OffsetHigh* в структуре *WSAOVERLAPPED* SPI-клиента. Когда вызывается функция *WSPGetOverlappedResult*, ваш поставщик должен сначала проверить поле *Internal* структуры *WSAOVERLAPPED* SPI-клиента. Если это поле содержит значение *WSS\_OPERATION\_IN\_PROGRESS*, поставщик все еще обрабатывает перекрытый запрос. Если значение *parametpafWait* в *WSPGetOverlappedResult* — *TRUE*, поставщик должен ждать завершения перекрытой операции: передачи описателя события в структуру *WSAOVERLAPPED* SPI-клиента перед возвращением результатов. При значении *FALSE* поставщик вернет ошибку *WSA\_LOJNCOMPLETE*. Как только перекрытая операция завершится, поставщик должен присвоить значения параметрам *WSPGetOverlappedResult*:

**III *lpcbTransfer*** — значение поля *InternalHigh* структуры *WSAOVERLAPPED*, сообщающее о количестве переданных операцией отправки или получения байтов;

- ***lpdwFlags*** — значение поля *Offset* структуры *WSAOVERLAPPED*, сообщающее о флагах, являющихся результатом операций *WSPRecv* или *WSPRecvFrom*;



Я *ipErrno* — значение поля *OffsetHigh* структуры *WSAOVERLAPPED*, сообщающее о результирующем коде ошибки.

Листинг 14-3 демонстрирует один из способов реализации *WSPGetOverlappedResult*.

### Листинг 14-3. Подробности реализации функции *WSPGetOverlappedResult*

```

BOOL WINAPI WSPGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno)
{
    DWORD Ret;

    if (lpOverlapped->Internal != WSS_OPERATION_IN_PROGRESS)
    {
        •lpcbTransfer = lpOverlapped->InternalHigh;
        •lpdwFlags = lpOverlapped->Offset;
        •IpErrno = lpOverlapped->OffsetHigh;

        return(lpOverlapped->OffsetHigh == 0 ? TRUE : FALSE);
    }
    else
    {
        if (fWait)
        {
            Ret = WaitForSingleObject(lpOverlapped->hEvent,
                INFINITE);
            if ((Ret == WAIT_OBJECT_0) &&
                (lpOverlapped->Internal !=
                 WSS_OPERATION_IN_PROGRESS))
            {
                •lpcbTransfer = lpOverlapped->InternalHigh;
                •lpdwFlags = lpOverlapped->Offset;
                •IpErrno = lpOverlapped->OffsetHigh;

                return(lpOverlapped->OffsetHigh == 0 ? TRUE :
                    FALSE);
            }
            else
                •IpErrno = WSASYSCALLFAILURE;
        }
        >
        else
            •IpErrno = WSA_IO_INCOMPLETE;
    }

    return FALSE;
}

```

**Процедуры завершения.** Завершая основанный на процедурах перекрытый ввод-вывод, SPI-клиент вызывает одну из функций ввода-вывода со структурой *WSAOVERLAPPED* и указателем *WSAOVERLAPPED\_COMPLETION\_ROUTINE*. Ваш поставщик службы отвечает за управление запросом перекрытого ввода-вывода. Когда запрос завершается, поставщик должен уведомить поток вызывающей программы о завершении перекрытого ввода-вывода, используя Win32-Механизм *асинхронного вызова процедур* (asynchronous procedure call, APC). APC требует, чтобы поток вызывающей программы был в «тревожном» состоянии ожидания (см. главу 8). Заканчивая обслуживание завершения основанного на процедурах перекрытого запроса, поставщик должен уведомить SPI-клиента через перекрестный вызов функции *WPUQueueApc*.

```
int WPUQueueApc(
    LPWSATHREADID lpThreadId,
    LPWSAUSERAPC lpfnUserApc,
    DWORD dwContext,
    LPINT lpErrno
);
```

Параметр *lpThreadId* представляет структуру *WSATHREADID* SPI-клиента, передаваемую исходным вызовом ввода-вывода и обеспечивающую завершение процедуры. Параметр *lpfnUserApc* — указатель на промежуточную функцию *WSAUSERAPC*, которую поставщик должен предоставить для обратного вызова SPI-клиенту. Промежуточная функция вызывает *WSAOVERLAPPED\_COMPLETION\_ROUTINE* SPI-клиента, предоставленную исходным перекрытым вызовом. Прототип промежуточной функции *WSAUSERAPC* определен так.

```
typedef void (CALLBACK FAR * LPWSAUSERAPC)(DWORD dwContext);
```

Заметьте, что это определение функции содержит только один параметр — *dwContext*. Когда SPI-клиент вызывает эту функцию, *dwContext* содержит информацию, первоначально переданную в параметре *dwContext* в *WPUQueueApc*. По существу, *dwContext* позволяет передавать структуру данных, содержащую любые нужные элементы информации, при вызове *WSAOVERLAPPED\_COMPLETION\_ROUTINE* (см. главу 8):

```
void CALLBACK CompletionROUTINE(
    IN DWORD dwError,
    IN DWORD cbTransferred,
    IN LPWSAOVERLAPPED lpOverlapped,
    IN DWORD dwFlags
```

Ваш поставщик передает через параметр *dwContext* функции *WPUQueueApc* следующую информацию:

- Ж состояние перекрытой операции как код ошибки Winsock;
- число переданных через перекрытую операцию байт;
- структуру *WSAOVERLAPPED* вызывающей программы;
- флаги вызывающей программы, переданные вызову ввода-вывода.

Теперь поставщик может успешно вызвать *WSAOVERLAPPEDJCOMPLETION ROUTINE* клиента SPI из промежуточной процедуры завершения.

**Порты завершения.** Модель ввода-вывода порта завершения Winsock 2 реализована в модуле *Ws2\_32.dll*. Модель порта завершения основана на модели перекрытого ввода-вывода, организованной на процедурах. Поэтому, чтобы ваш поставщик службы управлял моделью порта завершения, не требуется никаких дополнительных указаний.

**Управление перекрытым вводом-выводом.** Когда SPI-клиент вызывает ваш многоуровневый поставщик, использующий любую из описанных моделей перекрытого ввода-вывода, администратор перекрытия поставщика в свою очередь вызывает нижестоящий поставщик, который использует метод перекрытого ввода-вывода, основанный на событиях или на порте завершения. Если ваш поставщик выполняется под управлением Windows NT или Windows 2000, мы рекомендуем использовать порты завершения для перекрытого ввода-вывода на нижестоящем поставщике. В Windows 95 и Windows 98 следует использовать только основанный на событиях перекрытый ввод-вывод. Для работы с перекрытыми событиями вашему поставщику доступны следующие три вызова:

```
WSAEVENT WPUCreateEvent(LPINT lpErrno);  
BOOL WPUResetEvent(WSAEVENT hEvent, LPINT lpErrno);  
BOOL WPUCloseEvent(WSAEVENT hEvent, LPINT lpErrno);
```

Функция *WPUCreateEvent* создает и возвращает объект события, которое находится в режиме сброса вручную — точное подобие функции *WSACreateEvent* (см. главу 8). Если *WPUCreateEvent* не сможет создать объект события, возвращается *NULL*, а параметр *lpErrno* будет содержать определенный код ошибки Winsock. Функция *WSPResetEvent*, точно так же, как *WSAResetEvent*, переводит объект события (параметр *hEvent*) из незанятого в занятое состояние. Функция *WPUCloseEvent* — аналог *WSACloseEvent*, она освобождает все рабочие ресурсы, связанные с описателем объекта события.

В примере LSP на прилагаемом компакт-диске основанный на событиях перекрытый ввод-вывод используется для управления всеми действиями перекрытого ввода-вывода для SPI-клиента. Это позволяет примеру функционировать под Windows 2000, Windows NT, Windows 98 и Windows 95. Важно, что в этом примере операции перекрытого ввода-вывода обслуживает только один поток, что ограничивает возможности поставщика: он может обслуживать одновременно не более чем *WSA\_MAXIMUM\_WAIT\_EVENTS* (64) объектов событий.

Чтобы преодолеть этот барьер, создайте несколько потоков обслуживания. А еще лучше (особенно если вы разрабатываете поставщик для Windows NT и Windows 2000) — используйте вместо основанного на событиях перекрытого ввода-вывода порты завершения.

## Расширенные функции

Библиотека Winsock *Mswsock.lib* обеспечивает приложения расширенными функциями, увеличивающими возможности Winsock. В настоящее время поддерживаются следующие расширенные функции:

- *AcceptEx* — *WSAID\_ACCEPTEX*;

III *GetAcceptExSockaddrs* — *WSAID\_GETACCEPTEXSOCKADDRS*\

№ *TransmitFile* — *WSAIDJTRANSMITFILE*;

III *WSARecvEx* — не связана с GUID.

Когда приложение, связанное с *Mswsock.lib*, использует *AcceptEx*, *GetAcceptExSockaddrs* и *TransmitFile*, оно неявно вызывает функцию *WSPIoctl* вашего поставщика, используя *SIOjGET\_EXTENSION\_FUNCTION\_POINTER*. Функция *WSPIoctl* определена так:

```
int WSPIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,
    LPINT lpErrno
```

Когда происходит вызов, параметру *dwIoControlCode* присваивается значение *SIO\_GET\_EXTENSIONJUNCTION\_POINTER*. Параметр *lpvInBuffer* содержит указатель на GUID, который определяет расширенную функцию для нужд *Mswsock.lib* (значения GUID, перечисленные в списке, определяют поддерживаемые в настоящее время расширенные функции). Если это значение GUID соответствует любому из определенных значений, ваш поставщик должен вернуть через *lpvOutBuffer* указатель на реализацию этой расширенной функции. Остальные параметры для управления расширенными функциями в SPI прямо не используются.

Мы отметили, что функция *WSARecvEx* не имеет GUID. Дело в том, что *WSARecvEx* вызывает не *WSPIoctl*, а прямо *WSARecv*. В результате, поставщик не может напрямую контролировать расширенную функцию *WSARecvEx*.

## Установка поставщиков транспортной службы

Установка поставщиков транспортной службы включает разработку простого приложения для вставки многоуровневого или базового поставщика в каталог поставщиков служб Winsock 2. Многоуровневый и базовый поставщики транспорта устанавливаются по-разному. Программа установки просто настраивает поставщик в базе данных системной конфигурации Winsock 2, которая является каталогом всех установленных поставщиков служб. База данных конфигурации дает Winsock 2 знать, что поставщик существует, и определяет тип его службы. Winsock 2 использует эту БД, чтобы определить, какие поставщики службы следует загрузить при создании сокета приложением Winsock. *Ws2\_32.dll* ищет в базе данных первый поставщик, соответ-

ствующий сокетным входным параметрам вызовов *socket* и *WSASocket* (таким как семейство адреса, тип сокета и протокол) Как только подходящая запись найдена, *Ws2\_32.dll* загружает соответствующую DLL-библиотеку поставщика службы, определенную в каталоге

По существу, для успешных установки и управления записью поставщика службы в БД поставщиков необходимы четыре функции конфигурации SPI Каждая функция начинается с префикса *WSC* *WSCEnumProtocols*, *WSCInstallProvider*, *WSCWnteProviderOrder*, *WSCDeinstallProvider*

Эти функции работают с базой данных, используя структуру *WSAPROTOCOLINFO* (см главу 5) При установке поставщика транспортной службы нас прежде всего интересуют поля *ProviderId*, *dwCatalogEntryId* и *ProtocolChain* этой структуры Поле *ProviderId* — GUID, позволяющий уникально определять и устанавливать поставщик в любой системе Поле *dwCatalogEntryId* просто определяет каждую структуру *WSAPROTOCOLINFO* записи каталога в БД по уникальному числовому значению Поле *ProtocolChain* определяет, является ли структура *WSAPROTOCOLINFO* записью каталога для базового, многоуровневого или цепочки протоколов поставщика Поле *ProtocolChain* — структура *WSAPROTOCOLCHAIN*

```
typedef struct {
    int ChainLen,
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN],
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

Поле *ChainLen* определяет, представляет ли запись каталога многоуровневый (*ChainLen* = 0), основной (*ChainLen* = 1) поставщик, или цепочку протоколов поставщика (*ChainLen* > 1) Цепочка протоколов (protocol chain) — специальная запись каталога, задающая, как поместить многоуровневый поставщик служб между Winsock и другими поставщиками службы (рис 14-2) Многоуровневые и основные поставщики имеют только по одной записи каталога в БД Последнее поле — *ChainEntries*, представляет собой массив идентификаторов каталога, используемых для описания порядка загрузки поставщиков в цепочке протоколов Когда в ходе создания сокета *Ws2\_32.dll* ищет в каталоге соответствующий поставщик, она просматривает только записи цепочек протоколов и базовых поставщиков Записи многоуровневых поставщиков (*ChainLen* - 0) игнорируются — они существуют только для сопоставления многоуровневого поставщика цепочке протоколов в записях этих цепочек.

## Установка базового поставщика

Чтобы установить основной поставщик, создайте структуру *WSAPROTOCOLINFO* записи каталога, которая представляет этот поставщик Заполните поля в этой структуре информацией об атрибутах протокола Не забудьте присвоить значение 1 полю *ChainLen* структуры *ProtocolChain* Когда структура определена, поместите ее в каталог, используя функцию *WSCInstallProvider*

```
int WSCInstallProvider(
    const LPGUID lpProviderId,
    const LPWSTR lpszProviderDllPath,
```

```
const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
DWORD dwNumberOfEntries,
LPINT lpErrno
).
```

Параметр *ipProviderId* — GUID, позволяющий идентифицировать поставщик для каталога Winsock. Параметр *ipProviderDllPath* — строка, содержащая загрузочный путь к DLL поставщика. Строка может включать переменные окружения, такие как %SystemRoot%. Параметр *lpProtocolInfoList* представляет массив структур данных *WSAPROTOCOL\_INFOW* для помещения в каталог. Устанавливая основного поставщика, вы можете просто назначить структуре *WSAPROTOCOL\_INFOW* первый элемент массива. Параметр *dwNumberOfEntries* содержит количество записей в массиве *lpProtocolInfoList*, параметр *lpErrno* — код ошибки, если выполнение этой функции не удалось. Тогда возвращается *SOCKETERROR*.

### Установка многоуровневого поставщика

Чтобы установить многоуровневый поставщик службы, создайте две структуры *WSAPROTOCOL\_INFOW* записей в каталоге. Одна будет представлять ваш многоуровневый поставщик (например, длина цепочки протоколов, равная 0), а другая — цепочку протоколов (например, длина цепочки протоколов, большая 1), соединяющую многоуровневый поставщик с основным. Эти две структуры следует инициализировать со свойствами структуры *WSAPROTOCOL\_INFOW* записи каталога существующего поставщика службы. Запись вы можете найти с помощью функции *WSCEnumProtocols*.

```
int WSCEnumProtocols(
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFOW lpProtocolBuffer,
    LPDWORD lpdwBufferLength,
    LPINT lpErrno
),
```

г( > v

Параметр *lpiProtocols* — необязательный массив значений. Если *lpiProtocols* содержит *NULL*, возвращается информация по всем доступным протоколам, если нет — только по протоколам, перечисленным в массиве. Параметр *lpProtocolBuffer* — предоставленный приложением буфер, заполненный структурами *WSAPROTOCOL\_INFOW* из каталога Winsock 2. Входящий параметр *lpdwBufferLength* содержит количество байт в переданном функцией *WSCEnumProtocols* буфере *lpProtocolBuffer*. При выходе в этот параметр заносится минимальный размер буфера, который можно передать *WSCEnumProtocols* для определения всей запрошенной информации. Параметр *lpErrno* содержит информацию об ошибке, если выполнение функции неудачно и возвращено *SOCKET\_ERROR*. Располагая записью в каталоге для поставщика, который вы собираетесь поместить сверху, скопируйте свойства структуры *WSAPROTOCOL\_INFOW* поставщика, во вновь созданные структуры.

После инициализации поместите в каталог запись многоуровневого поставщика, используя функцию *WSCInstallProvider*. Затем найдите идентифи-

катор каталога, который присваивается этой структуре после установки, перебрав записи в каталоге с помощью *WSEnumProtocols*. Эта запись может затем использоваться при помещении в каталог записи цепочки протоколов, связывающей ваш многоуровневый поставщик с другим поставщиком. Затем для установки включенного в цепочку поставщика вызывается функция *WSCInstallProvider*. Этот процесс иллюстрирует следующий псевдокод:

```

WSAPROTOCOL_INFO LayeredProtocolInfoBuff,
                  ProtocolChainProtoInfo,
                  BaseProtocolInfoBuff;

// Получение BaseProtocolInfoBuff с помощью WSEnumProtocolsO

memcpy (&LayeredProtocolInfoBuff, &BaseProtocolInfoBuff,
        sizeof(WSAPROTOCOL_INFO));
LayeredProtocolInfoBuff.dwProviderFlags = PFL_HIDDEN;
LayeredProtocolInfoBuff.ProviderId = LayeredProviderGuid;

// Эта запись будет заполнена системой
LayeredProtocolInfoBuff.dwCatalogEntryId = 0;

LayeredProtocolInfoBuff.ProtocolChain.ChainLen =
    LAYERED_PROTOCOL;

WSCInstallProvider(&LayeredProviderGuid, L"lsp.dll",
                  JLayeredProtocolInfoBuff, 1, &install error);

// Определение идентификатора каталога многоуровневого поставщика с
// использованием функции WSEnumProtocolsO
for (i=0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
                sizeof (GUID))==0)
    {
        LayeredCatalogId = ProtocolInfo[i].dwCatalogEntryId;
        break;
    }

Memcpy(&protocolChainProtoInfo, &BaseProtocolInfoBuff,
        sizeof(WSAPROTOCOL_INFO));
ProtocolChainProtoInfo.ProtocolChain.ChainLen = 2;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[0] =
    LayeredProvideProtocolInfo.dwCatalogEntryId;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[1] =
    BaseProtocolInfoBuff.dwCatalogEntryId;

WSCInstallProvider(
    iChainedProviderGuid,
    L"lsp.dll",           // lpszProviderDHPPath
    &ProtocolChainProtoInfo, // lpProtocolInfoList
    tot,                  // tot
    ' ',                  // ' '
    <\,                    // <\
    r-,                    // r-
    . ^                    // . ^

```

```
1,                // dwNumberOfEntries
&install error    // lpErrno
```

Обратите внимание на флаг *PFLHIDDEN* в структуре *WSAPROTOCOLINFO*. Благодаря ему функция *WSAEnumProtocols* (см. главу 5) не включает каталог для многоуровневого поставщика в возвращаемый ею буфер.

Другой важный флаг, которым должна управлять программа установки — *XPIIFS\_HANDLES*. Любой не-IFS-поставщик службы, использующий *WPUCreateSocketHandle* для создания своих описателей сокета, не должен задавать флаг *XPIIFS\_HANDLES* в структуре *WSAPROTOCOLINFO*. Для приложений Winsock отсутствие флага *XPIIFS\_HANDLES* — указание избегать использования функций *ReadFile* и *WriteFile* из-за потенциального снижения производительности.

### Упорядочение поставщиков

Теперь вы должны решить, как Winsock 2 будет искать поставщиков служб в БД. Большинство приложений Winsock определяют нужный протокол с помощью параметров вызова функций *socket* и *WSASocket*. Например, если приложение создает сокет, используя семейство адресов *AF\_INET* и тип *SOCK\_STREAM*, Winsock 2 ищет заданный по умолчанию протокол TCP/IP, запись включенного в цепочку или базового поставщика в БД. Когда вы устанавливаете поставщик службы, используя *WSCInstallProvider*, запись в каталоге автоматически становится последней в БД. Чтобы сделать поставщик службы стандартным поставщиком TCP/IP, упорядочите записи о поставщиках в базе данных и поместите запись цепочки протоколов перед другими поставщиками TCP/IP. Для этого вызовите функцию *WSCWriteProviderOrder*.

```
int WSCWriteProviderOrder(
    LPDWORD lpwdCatalogEntryId,
    DWORD dwNumberOfEntries
```

Параметр *lpwdCatalogEntryId* принимает массив идентификаторов каталога, определяющих порядок его сортировки. Вы можете задать идентификаторы каталога, вызвав *WSEnumProtocols*, как описано ранее. Параметр *dwNumberOfEntries* — счетчик записей каталога в массиве. Эта функция возвращает *ERROR\_SUCCESS* (0), если выполнена успешно, и код ошибки Winsock — если нет.

Функция *WSCWriteProviderOrder* не входит в библиотеку *Ws2\_32.dll*. Чтобы использовать ее, приложение должно быть скомпоновано с библиотекой *Sporder.lib*. Модуль *Sporder.dll* не является частью ОС Windows, ищите DLL поддержки в библиотеке Microsoft Developer Network (MSDN). Если вы используете этот модуль, распространяйте его вместе с приложением.

Библиотека MSDN также предлагает удобную программную утилиту *Sporder.exe*, которая позволяет просматривать и переупорядочивать записи каталога в БД Winsock 2 (рис. 14-3)-



ServiceProviders | name Resolution

```
Layered TCP/IP over [MSAFD Tcpip [TCP/IP])
MSAFD Tcpip [TCP/IP]
MSAFD Tcpip [UDP/IP]
MSAFD Tcpip [RAW/IP]
RSVP UDP Service Provider
MSAFD NetBIOS (\Device\NetBT_Tcpip_{28013CC0-D0AA-11D2-8A53-8EE7542C000E})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{28013CC0-D0AA-11D2-8A53-8EE7542C000E})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{28013CC2-D1A-11D2-8A53-8EE7542C000E})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{28013CC2-D0AA-11D2-8A53-8EE7542C000E})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213832-D0B-11D2-8A53-00105A24E14A})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213832-D0B-11D2-8A53-00105A24E14A})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213831-D0B-11D2-8A53-00105A24E14A})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213831-D0B-11D2-8A53-00105A24E14A})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213830-D0B-11D2-8A53-00105A24E14A})
MSAFD NetBIOS (\Device\NetBT_Tcpip_{49213830-D0B-11D2-8A53-00105A24E14A})
```

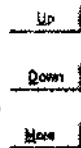


Рис. 14-3. Конфигурация Winsock 2 после установки многоуровневого поставщика на компьютер с Windows 2000

### Удаление поставщика службы

Удалить поставщик службы из каталога Wmsock 2 не сложно. Вызовите функцию *WSCDeinstallProvider*

```
int WSCDeinstallProvider(
    LPGUID lpProviderId,
    LPINT lpErrno
```

Параметр *lpProviderId* представляет GUID удаляемого поставщика службы. Параметр *lpErrno* получает код ошибки Wmsock, если функция возвращает *SOCKET\_ERROR*.

При удалении поставщика службы вы должны учитывать один важный момент. Всегда есть возможность, что многоуровневый поставщик службы включил идентификатор каталога вашего поставщика службы в свою цепочку протокола. Если это так, вы должны удалить идентификатор каталога из любых цепочек протоколов, которые ссылаются на ваш поставщик.

### Проблемы при установке многоуровневых поставщиков

Многоуровневые поставщики службы имеют огромный потенциал. Однако нынешняя спецификация Winsock 2 не отвечает на важный вопрос: как многоуровневый поставщик службы может узнать, в каком месте вставить себя в цепочку протоколов, если находит другой многоуровневый поставщик.

Например, если вы хотите установить поставщик шифрования данных в системе, которая уже содержит поставщик фильтрации URL, оче-

видно, что первый должен быть вставлен ниже второго в существующей цепочке протоколов. Но проблема в том, что программа установщика поставщика не может выяснить, какую службу обеспечивает существующий поставщик, и поэтому не знает, куда вставить новый. Это не приносит больших осложнений для управляемой сетевой среды, в которой администраторы решают, какие поставщики и в каком порядке устанавливать. Но широкому распространению многоуровневых поставщиков службы препятствует то, что безопасной может быть только установка многоуровневого поставщика службы непосредственно поверх основного и задание новой цепочки поставщиком протокола по умолчанию. Такой подход гарантирует службу нового поставщика, но удаляет существующий многоуровневый поставщик как заданную по умолчанию цепочку поставщиков.

Есть еще одна проблема, которую не разрешает спецификация Winsock 2. Она связана с первой, хотя и не столь серьезна. Как существующие многоуровневые поставщики могут защитить себя от изменений в цепочке или узнать, что такие изменения происходят? Если цепочка протоколов многоуровневого поставщика не должна меняться, его разработчик может жестко запрограммировать порядок цепочки в рамках функции *WSPStartup* и определить поставщик как базовый, указав значение 1 для параметра *ProtocolChmn ChamLen* структуры *WSAP-ROTOCOLINFO* для LSP.

## Поставщики службы пространства имен

В главе 10 мы рассмотрели, как приложение регистрирует и разрешает службы в пространстве имен, что особенно важно для служб, которые можно динамически создавать в сети. К сожалению, полезность существующих доступных пространств имен ограничена. Спецификация Winsock 2, однако, предлагает вам метод создания собственных пространств имени, где вы можете регистрировать и разрешать имена любым удобным для себя способом.

Это можно сделать с помощью DLL, реализующей девять функций пространства имен. Все функции начинаются с префикса NSP и соответствуют функциям RNR (см главу 10). Например, эквивалент функции пространства имени *WSASetService* — функция *NSPSetService*. После создания DLL она устанавливается в системный каталог с GUID, идентифицирующим пространство имен. Затем приложения могут регистрировать и запрашивать службы в вашем пространстве имен.

Сначала мы расскажем, как установить поставщик пространства имен, а затем опишем функции, которые он должен реализовать. Наконец, рассмотрим типовой поставщик пространства имен, а также типовое приложение, регистрирующее и разрешающее службы.

### Установка поставщика пространства имен

Поставщик пространства имен — это просто DLL, осуществляющая функции поставщика пространства имен. Чтобы приложения могли использовать

пространство имен, уведомите о нем систему с помощью функции *WSCInstallNameSpace*. Установленный поставщик вы можете отключить или удалить из системного каталога с помощью функций *WSAEnableNSProvider* и *WSAUninstallNameSpace*.

### Функция *WSCInstallNameSpace*

Эта функция используется для **установки поставщика** в системный **каталог** j  
Она определена так: >

```
int WSCInstallNameSpace (
    LPWSIRlpzIdentifier,
    LPWSIRlpzPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

t

Сразу бросается в глаза, что все строковые параметры представлены широкими символами. На самом деле, с использованием строк широких символов реализованы все поставщики пространства имен.

Параметр *lpzIdentifier* — имя поставщика пространства имен, возвращаемое функцией *WSAEnumNameSpaceProviders* (см. главу 10). Параметр *lpzPathName* — расположение DLL. Строка может включать системные переменные, такие как *%SystemRoot%*. Параметр *dwNameSpace* — числовой идентификатор пространства имен. Например, в заголовочном файле *Nsapi.h* определены константы других известных пространств имен, типа *NS\_SAP*, для IPX SAP. Параметр *dwVersion* задает номер версии пространства имен. Наконец, параметр *lpProviderId* — GUID, идентифицирующий поставщик пространства имен.

В случае удачного выполнения *WSCInstallNameSpace* возвращает 0, иначе — *SOCKET\_ERROR*. Чаще всего это *WSAEINVAL* (пространство имен с данным GUID уже существует) или *WSAEACCESS* (процесс вызова не имеет достаточных полномочий). Задавать пространство имен могут только члены группы Administrators.

### Функция *WSAEnableNSProvider*

Эта функция используется для изменения состояния (включения **или** выключения) поставщика пространства имен:

```
int WSAEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

Параметр *lpProviderId* — идентификатор GUID пространства имен, которое вы хотите изменить. Параметр *fEnable* содержит булево значение, указывающее, следует ли включить или выключить поставщик. Выключенный поставщик не может обрабатывать запросы или регистрацию.

При удачном выполнении *WSCEnableNSProvider* возвращает 0, иначе — *SOCKET\_ERROR*. Если GUID поставщика не действителен, возвращается ошибка *WSAEINVAL*.

### Функция *WSCUninstallNameSpace*

Эта функция удаляет поставщик пространства имен из каталога:

```
int WSCUninstallNameSpace ( LPGUID lpProviderId );
```

Параметр *lpProviderId* — GUID удаляемого пространства имен. Если GUID не действителен, выполнение возвращается ошибка *WSAEINVAL*.

## Реализация пространства имен

Пространство имен должно реализовать все девять функций, соответствующих функциям RNR. Кроме того, следует разработать метод для постоянного сохранения данных, то есть поддерживать данные вне экземпляра DLL. Каждый загружающий DLL процесс получает свой собственный сегмент данных — это означает, что сохраненные в рамках DLL данные не могут быть разделены между экземплярами (на самом деле, совместное использование информации загрузившими DLL приложениями возможно, но не одобрено).

Дополнительную информацию по библиотекам DLL можно найти в книге «Windows для профессионалов» Джеффри Рихтера (М.: «Русская Редакция», 2001). Как упоминалось в главе 10, существуют три типа пространства имен: динамическое, постоянное и статическое. Очевидно, реализовать статическое пространство не стоит, так как оно не допускает программную регистрацию служб. Далее мы рассмотрим некоторые аспекты поддержки данных, которые пространство имен должно сохранить.

Очень важно использовать строки широких символов во всех функциях поставщика пространства имен: не только строковых параметров функций, но также и строк в RNR-структурах, типа *WSAQUERYSET* и *WSASERVICECLASS-INFO*. Вы можете спросить, возможно ли это — ведь когда приложение регистрирует или разрешает имя, оно вправе использовать и обычную версию (ASCII), и версию с широкими символами (UNICODE) функций и RNR-структур. На самом деле, работает любая версия, так как все запросы ASCII проходят промежуточный уровень, преобразующий все строки в строки широких символов.

Это верно и для вызова функции, и для ее возвращения. То есть если вызывающему приложению возвращается *WSAQUERYSET* (с помощью функции *WSALookupServiceNext*), то любые данные, возвращенные поставщиком пространства имен, изначально находятся в виде UNICODE и преобразуются в ASCII перед возвратом из функции. Так что если ваши приложения используют RNR-функции, вызов версий с широкими символами будет быстрее, так как не требует никаких преобразований.

Из девяти функций, которые должен реализовать поставщик пространства имен, только семь соответствуют RNR-функциям Winsock 2 (табл. 14-2).

**Табл. 14-2. Соответствие функций регистрации и разрешения имен в Winsock 2 функциям поставщика пространства имен**

<b>функция Winsock</b>	<b>Эквивалентная функция поставщика пространства имен</b>
<i>WSAInstallServiceClass</i>	<i>NSPInstallServiceClass</i>
<i>WSARemoveServiceClass</i>	<i>NSPRemoveServiceClass</i>
<i>WSAGetServiceClassInfo</i>	<i>NSPGetServiceClassInfo</i> ! Т*
<i>WSASetService</i>	<i>NSPSetService</i> }\$1
<i>WSALookupServiceBegin</i>	<i>NSPLookupServiceBegin</i>
<i>WSALookupServiceNext</i>	<i>NSPLookupServiceNext</i> ЭГ
<i>WSALookupServiceEnd</i>	<i>NSPLookupServiceEnd</i>

Оставшиеся две функции применяются для инициализации и очистки. Когда пространство имен указано в системе, приложения могут использовать его, определяя соответствующий GUID или идентификатор пространства имен. Тогда приложение вызывает стандартную RNR-функцию Winsock 2, как описано в главе 10. При этом вызывается эквивалентная функция поставщика пространства имен. Например, при вызове из приложения функции *WSAInstallServiceClass*, которая ссылается на GUID пользовательского пространства имен, вызывается функция *NSPInstallServiceClass* для этого поставщика.

Далее мы рассмотрим все функции пространства имен.

**Функция NSPStartup**

*NSPStartup* вызывается каждый раз, когда загружается DLL поставщика пространства имен. Ваша реализация пространства имен должна включать эту функцию, экспортируемую из DLL. При этом можно выделить для каждой DLL все структуры данных, которые требуются для работы поставщика. Прототип *NSPStartup* выглядит так:

```
int NSPStartup (
    LPGUID lpProviderId,
    LPNSP_ROUTINE lpnspRoutines
);
```

Первый параметр — *lpProviderId*, является GUID для этого поставщика пространства имен. Параметр *lpnspRoutines* — структура *NSP\_ROUTINE*, которую должна заполнить ваша реализация этой функции. Эта структура обеспечивает указатели функции для восьми других функций пространства имен поставщика. Объект *NSP\_ROUTINE* определен так:

```
typedef struct _NSP_ROUTINE
{
    DWORD          cbSize;
    DWORD          dwMajorVersion;
    DWORD          dwMinorVersion;
    LPNSPCLEANUP   NSPCleanup;
    LPNSPLOOKUPSERVICEBEGIN NSPLookupServiceBegin;
    LPNSPLOOKUPSERVICENEXT  NSPLookupServiceNext;
    LPNSPLOOKUPSERVICEEND  NSPLookupServiceEnd;
```

```

LPNSPSETSERVICE      NSPSetService;
LPNSPINSTALLSERVICECLASS NSPInstallServiceClass;
LPNSPREMOVESERVICECLASS NSPRemoveServiceClass;
LPNSPGETSERVICECLASSINFO NSPGetServiceClassInfo;
} NSP_ROUTINE, FAR * LPNSP_ROUTINE;

```

Первое поле структуры — *cbSize*, указывает размер структуры *NSP\_ROUTINE*. Следующие два поля — *dwMajorVersion* и *dwMinorVersion*, включены для управления версиями поставщика. Управление версиями произвольно и не служит никакой другой цели. Поставщик присваивает остальным записям соответствующие указатели на функции. Например, свой адрес функции *NSPSetService* (независимо от того, что скрывается под этим именем) — *NSPSetService*. Имена функций поставщика могут быть произвольными, но их параметры и возвращаемые типы должны соответствовать определению поставщика.

Единственное действие, требуемое *NSPStartup*<sup>^</sup> — заполнить структуру *NSP\_ROUTINE*. Когда поставщик успешно завершает эту и любые другие процедуры инициализации, он возвращает *NO\_ERROR*. Если произошла ошибка, *NSPStartup* возвращает *SOCKETJERROR* и выясняет код ошибки Winsock. Например, если ошибка связана с распределением памяти, поставщик вызывает *WSASetLastError* с параметром *WSA\_NOT\_ENOUGH\_MEMORY*, а затем возвращает *SOCKET\_ERROR*.

Сейчас, наверное, самое время обсудить обработку ошибок в DLL поставщика. Все функции, которые вы должны реализовать для поставщика, возвращают *NO\_ERROR* в случае успеха и *SOCKETERROR* — при неудаче. Если вы видите, что вызов возвращает ошибку, назначьте соответствующий код ошибки Winsock. Иначе любое приложение, пытающееся регистрировать или вызывать службы, используя ваш поставщик пространства имен, сообщит о неудаче функции RNR, но *WSAGetLastError* вернет 0. Это вызовет проблемы у приложений, которые пытаются корректно обрабатывать ошибки, 0 — безусловно не то значение, которое ожидается при ошибке.

### Функция *NSPCleanup*

Эта процедура вызывается при выгрузке DLL поставщика. С ее помощью вы можете освобождать любую память, выделенную процедурой *NSPStartup*. Функция *NSPCleanup* определена так-

```
int NSPCleanup ( LPGUID lpProviderId );
```

Единственный параметр — GUID вашего поставщика пространства имен. От вас требуется лишь очистка всей динамически выделенной памяти.

### Функция *NSPInstallServiceClass*

Функция *NSPInstallServiceClass* соответствует *WSAInstallServiceClass* и отвечает за регистрацию класса служб.

```
int NSPInstallServiceClass (
    LPGUID lpProviderId,
```

```
LPWSASERVICECLASSINFOW IpServiceClassInfo
);
```

Первый параметр — GUID поставщика. Параметр *IpServiceClassInfo* — регистрируемая структура *WSASERVICECLASSINFOW*. Ваш поставщик должен поддерживать список классов служб и гарантировать, что не существует класса служб, использующего тот же самый GUID в структуре *WSASERVICECLASSINFOW*. Если же этот GUID уже используется, поставщик должен вернуть ошибку *WSAEALREADY*. Иначе ему придется поддерживать этот класс служб так, чтобы другие операции RNR могли ссылаться на него.

Большинство оставшихся функций поставщика пространства имен ссылаются на определенный класс служб.

### Функция *NSPRemoveServiceClass*

Эта функция дополняет *NSPInstallServiceClass* и соответствует *WSARemoveServiceClass*. Она удаляет указанный класс служб:

```
mt NSPRemoveServiceClass (
    LPGUID IpProviderId,
    LPGUID IpServiceClassId
);
```

Как и в предыдущей функции, первый параметр — GUID поставщика. Второй — *IpServiceClassId*, GUID удаляемого класса служб. Поставщик должен удалить данный класс служб из своей памяти. Если класс служб, указанный параметром *IpServiceClassId*, не найден, поставщик сгенерирует ошибку *WSATYPE\_NOT\_FOUND*.

### Функция *NSPGetServiceClassInfo*

Функция *NSPGetServiceClassInfo* соответствует *WSAGetServiceClassInfo*. Она извлекает связанную с GUID структуру *WSANAMESPACEINFOW*.

```
int NSPGetServiceClassInfo (
    LPGUID IpProviderId,
    LPWORD lpdwBufSize,
    LPWSASERVICECLASSINFOW IpServiceClassInfo
);
```

Опять первый параметр — GUID поставщика. Параметр *lpdwBufSize* указывает количество байт, содержащихся в третьем параметре — *IpServiceClassInfo*. Третий входной параметр — структура *WSASERVICECLASSINFOW*, содержит условия поиска, определяющие класс служб для возврата. В этой структуре могут находиться имя или GUID класса служб для возврата. Если поставщик обнаруживает соответствие условиям, он должен вернуть структуру *WSASERVICECLASSINFOW* в параметре *IpServiceClassInfo* и обновить параметр *lpdwBufSize*, указав количество возвращаемых байт.

Если не найдены искомые классы служб, будет выдана ошибка *WSATYPE\_NOT\_FOUND*. Если же соответствующий класс служб найден, но предоставленный буфер слишком мал, поставщик должен обновить параметр *lpdwBufSize*, указав правильное количество требуемых байт, и выдать ошибку *WSAEFAULT*.

### Функция *NSPSetService*

Функция *NSPSetService* соответствует *WSASetService* и также регистрирует или удаляет службы в пространстве имен:

```
int NSPSetService (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    LPWSAQUERYSETW lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

Первый параметр — GUID поставщика. Параметр *lpServiceClassInfo* — структура *WSASERVICECLASSINFOW*, к которой относится эта служба. Параметр *lpqsRegInfo* — служба, регистрируемая или удаляемая в зависимости от операции, указанной в четвертом параметре — *essOperation*. В последнем параметре — *dwControlFlags*, может быть определен флаг *SERVICE\_MULTIPLE*, изменяющий указанную операцию.

Поставщик пространства имен сначала проверяет, существует ли соответствующий класс служб. Далее, в зависимости от того, какая операция определена, предпринимается соответствующее действие (полное описание допустимых значений *essOperation* и эффекта *dwControlFlags* см. в разделе «Регистрация служб» главы 10). Функция *NSPSetService* поставщика соответственно обрабатывает эти флаги.

Если поставщик службы пытается обновить или удалить службу, которая не может быть найдена, вернется ошибка *WSASERVICE\_NOT\_FOUND*. Если поставщик регистрирует службу, а структура *WSAQUERYSETW* неверна или неполна, поставщик генерирует ошибку *WSAEINVAL*.

Эта функция, как и *NSPLookupServiceNext*, — одна из наиболее сложных для реализации поставщика пространства имен. Поставщик должен поддерживать схему сохранения служб, которые могут быть зарегистрированы и позволяют функции *NSPSetService* обновлять данные.

### Функция *NSPLookupServiceBegin*

Функция *NSPLookupServiceBegin* связана с функциями *NSPLookupServiceNext* и *NSPLookupServiceEnd* и соответствует *WSALookupServiceBegin*. Она используется для инициализации запроса пространства имен и задает условия поиска. Прототип функции выглядит так:

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
    LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

Первый параметр — GUID поставщика. Параметр *lpqsRestrictions* — структура *WSAQUERYSETW*, которая определяет параметры запроса. Третий пара-



метр — *ipServiceClassInfo*, является структурой *WSASERVICECLASSINFOW*, содержащей информацию о схеме запроса для класса служб. Параметр *dwControlFlags* может содержать флаги, влияющие на способ выполнения запроса (информацию по *WSALookupServiceBegin* и другим флагам см. в главе 10).

Заметьте, не все флаги имеют смысл для каждого поставщика. Например, если ваше пространство имен не поддерживает понятие контейнерных объектов, вас не должны беспокоить флаги, связанные с контейнерами (контейнер — это способ концептуальной организации служб, все, что составляет контейнер, открыто для интерпретации). Наконец, *iphLookup* — выходной параметр, который является описателем, определяющим этот запрос. Описатель используется в последующих запросах к *WSALookupServiceNext* и *WSALookupServiceEnd*.

При реализации *NSPLookupServiceBegin* имейте в виду, что эта операция не может быть отменена и должна завершиться так быстро, как возможно. Поэтому если вам нужно инициализировать сетевой запрос, для успешного возврата не требуйте ответа.

Сам поставщик должен сохранить параметры запроса и связать уникальный описатель с запросом, чтобы сослаться на него позже. Кроме того, поставщик обязан поддерживать информацию о состоянии.

### Функция *NSPLookupServiceNext*

Когда запрос инициализирован с помощью *WSALookupServiceBegin*, приложение вызывает *WSALookupServiceNext*, а она, в свою очередь, функцию *NSPLookupServiceNext* поставщика пространства имен. Этот вызов ищет результаты, соответствующие условиям поиска для данного запроса. Функция определена так:

```
int NSPAPI WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBuffertlength,
    LPWSAQUERYSET lpqsResults
).
```

Первый параметр — *hLookup*, является описателем запроса, возвращенным функцией *WSALookupServiceBegin*. Параметр *dwControlFlags* может быть флагом *LUP\_FLUSHPREVIOUS*, указывающим, что поставщик должен отказаться от последнего набора результатов и перейти к следующему. Обычно приложение делает это, когда не может предоставить достаточно большой буфер для результатов. Следующий параметр — *lpdwBufferLength*, указывает размер б>фера, переданного в последнем параметре — *lpqsResults*.

Когда вызвана функция *NSPLookupServiceNext*, поставщик должен найти параметры запроса, определенного описателем *hLookup*. Как только эти параметры найдены, инициализируется поиск всех зарегистрированных и соответствующих предоставленным условиям служб в пределах класса, указанного запросом. Как мы уже говорили, состояние запроса должно быть сохранено. Если найдено несколько соответствующих записей, процесс запроса

вызывает *WSALookupServiceNext* многократно, и с каждым вызовом поставщик должен вернуть набор данных

Когда соответствующих записей больше нет, поставщик возвращает ошибку *WSA\_E\_NO\_MORE*. Можно отменить запрос в процессе выполнения, если приложение обращается к *WSALookupServiceEnd* из другого потока, в то время как происходит вызов *WSALookupServiceNext*. В этом случае выполнение *NSPLookupServiceNext* не будет успешным и функция вернет ошибку *WSA\_E\_CANCELLED*.

## Функция *NSPLookupServiceEnd*

По завершении запроса вызывается функция *NSPLookupServiceEnd*, чтобы закончить запрос и освободить все базовые ресурсы

```
int NSPLookupServiceEnd ( HANDLE hLookup ),
```

Единственный параметр функции — *hLookup*, является описателем закрываемого запроса. Если данный описатель не может быть найден (например, он недействителен), возвращается ошибка *WSA\_INVALID\_HANDLE*.

## Пример

Мы шаг за шагом рассмотрели процесс создания собственного пространства имен и коснулись некоторых связанных с этим проблем, например, методов постоянного сохранения данных. Однако, разработка поставщика пространства имен может быть сложной, поэтому проиллюстрируем свой рассказ примером. Приведенный в примере поставщик не самый быстрый, и его код далеко не оптимален. Тем не менее, он иллюстрирует моменты, требующие пристального внимания и понять его довольно просто.

Пример расположен на прилагаемом к книге компакт-диске в каталоге *Examples\Chapter14\NSP* в файлах *Mynsph*, *Mynspcrr* и *Mynspdef*. Эти три файла составляют DLL пространства имени. Кроме того, вы найдете на компакт-диске службу пространства имен — сервер Winsock, ответственный за обработку запросов от DLL. Этот сервер, обслуживающий регистрационные данные службы, находится в файле *Mynspsvc.crr*. Два дополнительных файла — *Nspsvc.crr* и *Pnntob.crr*, используются и DLL, и службой, и содержат процедуры поддержки маршалинга и демаршалинга данных, пересылаемых через сокет между службой и DLL. Там же вы найдете их заголовочные файлы — *Nspsvc.h* и *Pnntob.h*, содержащие прототипы функций для процедур поддержки. Наконец, файл *Rnrsc* — измененный пример из главы 10, регистрирует и ищет службы в нашем пространстве имен.

В следующих разделах мы обсудим реализацию пространства имен. Сначала рассмотрим метод, выбранный для постоянного сохранения данных. Этот обзор будет сопровождаться изучением структуры DLL пространства имен, а также установки пространства имен. Далее рассмотрим реализацию службы пространства имен. Наконец, мы покажем, как приложение выполняет регистрацию и запросы к пользовательскому пространству имен.

## Сохранение данных

Для нашего пространства имен мы выбрали отдельную службу Winsock, обслуживающую информацию пространства имен. В каждой реализованной в DLL функции пространства имен осуществляется соединение с этой службой и передаются данные для завершения операции. Для простоты, эта служба выполняется локально (слушает на адресе обратной петли 127.0.0.1). В фактической реализации IP-адрес службы пространства имен будет доступен через системный реестр или некоторые другие средства. Так что если приложение вызовет пространство имен, оно сможет соединиться со службой везде, где бы ни выполнялось. Например, в случае DNS, IP-адрес DNS-сервера будет назначен статически либо получен в ходе запроса DHCP.

Конечно, написать службу не единственный выход: вы можете обслуживать файл, который сохраняет необходимую информацию, в сети. Но мы не советуем выбирать последний вариант, поскольку тогда производительность пространства имен будет ограничена дисковыми операциями. В нашем примере производительность ограничивает и то, что пространство имен устанавливает соединение со службой через TCP. Промышленная реализация, вероятно, будет использовать дейтаграммный протокол без установления соединения, такой как UDP. Конечно, это повлечет дополнительные программные требования (в частности, чтобы отброшенные пакеты передавались повторно), но производительность будет увеличена значительно.

## DLL пространства имен

Прежде чем говорить о том, как реализована служба пространства имен, давайте рассмотрим его DLL. Каждый поставщик пространства имен требует уникальный GUID; эти GUID определены в файле *Mynsp.h*. Кроме уникального идентификатора, нам нужен простой целочисленный идентификатор, который может использоваться в поле *dwNameSpace* структуры *WSAQUERYSET*. GUID и идентификатор пространства имен выглядят так:

```
GUID MY_NAMESPACE_GUIDO = {0x55a2bd9e, 0xb530, 0x11d2,
                             {0x91, 0x66, 0x00, 0xa0, 0xc9, 0xa7, 0x86, 0xe8}}
};

#define NS_MYNSP66
```

Эти значения важны, так как приложения, которые хотят использовать данное пространство имен, должны указывать их в своих запросах Winsock. Конечно, разработчик приложения может определить эти значения явно или извлечь их, вызвав *WSAEnumNameSpaceProviders* (см. главу 10).

Если приложение выполняет операцию, указывая поставщика имен *NS\_ALL*, операция происходит на всех имеющихся поставщиках имен. Некоторые приложения Windows, например, Microsoft Internet Explorer, выполняют запросы на всех поставщиках имен. Поэтому будьте предельно осторожны, тестируя поставщик имен: если он плохо написан, то может вызвать общесистемные проблемы. Кроме того, значения GUID и идентификатора пространства имен важны, так как требуются для установки поставщика имен.

Теперь давайте посмотрим на NSP-функции, реализованные в *Mynsp.cpp*. Почти все они очень похожи, кроме функций запуска и очистки — *NSPStartup*

и *NSPCleanup*. Функция запуска просто инициализирует структуру *NSP\_ROUTINE* пользовательскими функциями пространства имен. Процедура очистки не делает ничего, потому что никакая очистка не требуется.

Остальные функции требуют взаимодействия с нашей службой, чтобы сделать запрос или зарегистрировать данные. Чтобы установить связь со службой, выполните следующие шаги.

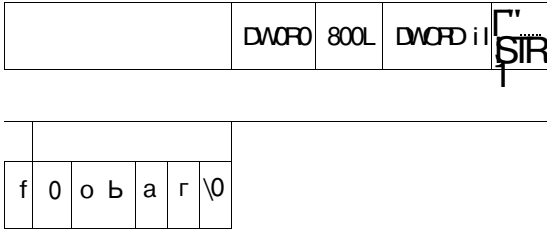
1. Соединитесь со службой (через функцию *MyNspConnect*).
2. Запишите однобайтный код действия. Он укажет службе, какое действие будет произведено (регистрация, удаление, запрос и т. п.).
- 3- Маршализируйте параметры и отправьте их службе. Тип параметров зависит от операции. Например, *NSPLookupServiceNext* отправляет описание запроса службе, чтобы та могла возобновить запрос, а *NSPSetService* — всю структуру *WSAQUERYSET*.
4. Прочитайте код возврата. Когда служба обладает необходимыми для выполнения требуемой операции параметрами, возвращается код возврата операции (успех или неудача). Файл *Mynsp.h* определяет для этой цели две константы — *MYNSP\_SUCCESS* и *MYNSP\_ERROR*.
5. Если требуемая операция была запросом, а код возврата свидетельствует об успешном выполнении, прочтите и демаршализируйте результаты. Например, *NSPLookupServiceNext* возвращает структуру *WSAQUERYSET*, когда найдена соответствующая служба.

Как видите, реализация DLL не слишком сложна. NSP-функции должны принимать параметры и обрабатывать их, что в нашем случае означает — передать эту информацию службе пространства имен. После этого выполнение запрошенной операции — дело службы. Однако мы пропустили одну трудную операцию, которая должна быть выполнена: отправка данных через сокет. Обычно для нее нет никаких особых требований, но при отправке целых структур данных они есть.

Большинство функций пространства имен принимают структуру *WSAQUERYSET* или *WSASERVICECLASSINFO* в качестве параметра. Объект должен быть отправлен или получен через сокетное соединение со службой. Это представляет некоторую трудность, так как данные структуры не являются непрерывными блоками памяти. Они содержат указатели на строки и другие структуры, которые могут быть расположены где-нибудь в памяти (рис. 14-4). Вы должны взять все эти части памяти — везде, где они есть — и скопировать в один буфер, одну за другой. Данный метод известен как *маршалинг данных* (marshaling data). На приемнике процесс должен быть повторен в обратном порядке: данные, которые нужно прочитать, будут вновь собраны в первоначальную структуру, а указатели — заданы так, чтобы ссылались на действительное расположение памяти на компьютере получателя.

Функции маршалинга и демаршалинга структур *WSANAMESPACEINFO* и *WSAQUERYSET* нашего поставщика пространства имен расположены в файле *Nspsvc.cpp* и используются как DLL пространства имен, так и службой пространства имен (так как обе стороны нуждаются в маршалинге и демаршалинге этих структур). Все четыре функции не требуют пояснений.

## WSASERVICECLASSINFO



## Marshaling

ИЛИ WSOCKET

Рис. 14-4. Маршалинг данных

## Установка и удаление поставщика пространства имен

Это самый простой шаг во всем процессе. Файл *Nspinstall.c* — простая программа установки. Следующий код устанавливает наш поставщик:

```
ret = WSCInstallNameSpace(L"Custom Name Space Provider",
    rXSystemRootX\System32\Mynsp.dll", NS_NAMESPACE, 1,
    &MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to install name space provider: %d\n",
        *WSAGetLastError());
}
```

Параметры вызова имя поставщика, расположение DLL, целочисленный идентификатор, версия и GUID. После установки нужно только удостовериться, что DLL пространства имен на самом деле расположена там, где вы указали. Единственная возможная ошибка — попытка установить поставщик имен с GUID, который уже используется другим поставщиком.

Удалить поставщик пространства имен еще проще

```
ret = WSCUnInstallNameSpace(m_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider: %d\n", WSAGetLastError());
}
```

## Служба пространства имен

Служба пространства имен — реальное содержание поставщика имен. Эта служба следит за всеми зарегистрированными классами и экземплярами служб. Когда DLL пространства имен вызвана приложением пользователя, она соединяется со службой пространства имен для выполнения операции. Служба проста. В функции *main* назначается прослушивающий сокет. Далее

в цикле принимаются соединения от экземпляров DLL пространства имен. Для простоты, в каждый момент обрабатывается только одно соединение. Это также избавляет от необходимости синхронизировать доступ к структурам данных, которые обслуживают информацию пространства имен. (Опять же, настоящий поставщик работал бы иначе, чтобы не снижать производительность.) Как только соединение принято, служба читает один байт из DLL пространства имен, идентифицирующий следующее действие.

В цикле действие декодируется, параметры передаются службе от DLL пространства имен. Затем выполняются требуемые действия. Они не сложны и исследуя их пошагово, вы сможете увидеть, как работает служба. Здесь мы не будем вдаваться в детали, исследуем лишь структуры, обслуживающие информацию.

Есть только два типа данных, к которым имеют отношение поставщики пространства имен: структуры *WSASERVICECLASSINFO* и *WSAQUERYSET*. Как вы видели, большинство RNR-функций ссылаются в своих параметрах на одну из этих структур. В результате, мы обслуживаем два глобальных массива — по одному для каждого типа структуры, а также счетчик для каждого из них.

Когда DLL запрашивает установку класса служб, функция *main* поставщика служб имен сначала вызывает *LookupServiceClass* — процедуру поддержки, определенную в *Mynspsvc.spp*. Эта функция просматривает массив *ServiceClasses* из структур *WSASERVICECLASSINFO*. Если найден класс служб с тем же самым GUID, служба возвращает ошибку (которую DLL транслирует, как *WSAEALREADY*). Иначе в конец массива добавляется новый класс служб, а счетчик *dwNumServiceClasses* увеличивается на 1.

При удалении класса служб (как и при установке) функция *main* вызывает *LookupServiceClass*. Если класс служб найден, код перемещает его на место удаленного класса и уменьшает счетчик на 1. В спецификации Winsock 2 для поставщиков пространства имен не указано, что происходит, когда класс служб должен быть удален, но есть ссылающиеся на него зарегистрированные службы. Как вы будете обрабатывать этот случай — ваше дело. Наше пространство имен в такой ситуации не допустит удаления класса службы.

Принцип, используемый для поддержания структур *WSASERVICECLASSINFO*, применяется и для отслеживания структур *WSAQUERYSET*. Существует массив этих структур с именем *Services* и счетчик с именем *dwNumServices*. Добавление и удаление служб выполняется так же, как и добавление и удаление классов.

Служба должна отвечать на запросы. Когда приложение инициализирует запрос, параметры запроса должны поддерживаться в течение его срока жизни и для них следует назначить уникальный описатель. Это необходимо, так как *WSALookupServiceNext* обращается к запросу только по описателю. Другая часть информации, которая должна сохраняться, — состояние запроса. Каждый запрос к *WSALookupServiceNext* возвращает уникальный набор информации. Код должен помнить последнюю позицию в массиве *Services*, для которой были возвращены данные, чтобы последующие запросы к *WSALookupServiceNext* начинали с того места, где остановился предыдущий.

## Запрос пространства имен

Последняя часть нашего пространства имен — файл `Rnrcs.c`. Это измененная версия примера регистрации и разрешения имен из главы 10. Мы внесли только несколько изменений, чтобы предельно упростить пример. Первое изменение заставляет код перечислять установленные поставщики пространства имен, но возвращать только поставщик `NSJMYNSP`. Во-вторых, при регистрации службы `Rnrcs.c` перечисляет только локальные интерфейсы IP для использования в качестве адреса службы. Наш поставщик службы поддерживает регистрацию любого типа `SOCKADDR`. Наконец, для регистрации службы этот пример не создает экземпляра службы, а только регистрирует имя.

## Выполнение примера

После компиляции всех примеров установка и использование поставщика просты. Следующая команда устанавливает поставщик:

```
Nspinstall.exe install
```

Конечно, не забудьте скопировать `Mynsp.dll` в `%SystemRoot%\System32`. Когда пространство имен установлено, должен быть запущен экземпляр службы, чтобы делать запросы и регистрировать службы. Это делается командой `Mynspsvc.exe`.

Теперь вы можете делать запросы и регистрировать службы, используя `Rnrcs.exe`. Перечислим некоторые команды, которые вы должны выполнить:

- **`Rnrcs.exe -s:ASERVICE`** — регистрация службы `ASERVICE`;
- И **`Rnrcs.exe -s:BSERVICE`** — регистрация службы `BSERVICE`;
- **`Rnrcs.exe -c:*`** — запрос всех зарегистрированных служб;
- **`Rnrcs.exe -c:BSERVICE`** — запрос только служб с именем `BSERVICE`;
- **`Rnrcs.exe -c:ASERVICE -d`** — запрос и удаление только служб с именем `ASERVICE`;
- **`Rnrcs.exe -c:BSERVICE -d`** — запрос и удаление только служб с именем `BSERVICE`;
- **`Rnrcs.exe -c:*`** — запрос всех зарегистрированных служб.

Эта последовательность команд регистрирует две службы и выполняет параметризованный и конкретный запрос. Затем последовательность команды делает запрос для каждого из этих двух служб и удаляет их. Наконец, мы выполняем параметризованный запрос, чтобы показать, что службы удалены.

## Отладочные функции отслеживания Winsock 2 SPI

Winsock 2 может отслеживать запросы API и SPI, вызванные из `Ws2_32.dll`. Это чрезвычайно полезно при разработке поставщика служб пространства имен или транспорта. В MSDN SDK два примера — `Dt_dll` и `Dt_dll2`, созданы для отслеживания вызовов SPI, когда они обрабатываются системой. Пре-

лесть этих примеров в том, что вы можете изменить их, чтобы отслеживать любые вызовы API и SPI, которые вас интересуют.

Чтобы использовать эту особенность отладки, получите сначала проверенную сборку Winsock 2 `Ws2_32.dll` для целевой платформы. Она содержится в MSDN SDK в папке `Mssdk\Bin\Debug\Winsock`. Вы можете скопировать эту сборку поверх системной библиотеки `Ws2_32.dll` в `%SystemRoot%\System32` или поместить ее в рабочий каталог своего приложения Winsock. Последнее лучше всего, так как вам не придется менять систему в целях отладки. После того как проверенная сборка установлена, скомпилируйте и соберите один из примеров MSDN `Dt_dll`. Теперь вы получили DLL поддержки с именем `Dt_dll.dll`. Скопируйте `Dt_dll.dll` в рабочий каталог своего приложения Winsock и можете отслеживать функции Winsock 2 API и SPI, которые косвенно вызываются из вашего приложения.

## Резюме

Winsock 2 SPI предлагает разработчикам метод расширения возможностей Winsock 2 с помощью разработки поставщика службы. В этой главе мы изложили подробности разработки поставщика транспорта и поставщика пространства имен.

Данная глава завершает обсуждение сетевой технологии Winsock. В следующей главе рассматривается элемент управления Microsoft Visual Basic Winsock, который использует Winsock. Мы не будем вводить никакие новые концепции, а просто дадим рекомендации по работе с этим элементом. Если вас не интересует Visual Basic, вы можете перейти к третьему, заключительному разделу книги, в котором рассматривается служба RAS — технология, которая позволяет повысить гибкость приложений Winsock.



## Элемент управления Winsock

Поставляемый с Microsoft Visual Basic элемент управления Winsock — относительно новый элемент управления, он превращает Winsock в удобный естественный интерфейс, доступный из Visual Basic. До его появления для сетевого программирования на Visual Basic с использованием Winsock приходилось импортировать все функции Winsock из DLL-библиотеки и переопределять множество необходимых структур. Этот процесс был длительным и чреват ошибками, например, из-за несоответствия объявлений типов. Тем не менее, если вам требуется дополнительная гибкость, обеспечиваемая непосредственным импортом Winsock в Visual Basic, изучите примеры программ на Visual Basic из второй части книги. Каждый пример включает файл Winsock.bas, импортирующий необходимые константы и ограничения.

В этой главе рассматривается только элемент управления Winsock из Visual Basic. Сначала мы обсудим свойства и методы Winsock, а затем приведем несколько примеров, в которых используется этот элемент управления.

Впервые Winsock появился в Visual Basic 5.0. Во второй и третий пакеты обновлений Visual Studio включена усовершенствованная версия данного элемента управления. С Visual Basic 6.0 поставляется новейшая версия Winsock. Мы обсудим различия этих версий.

Элемент управления Winsock реализует лишь базовый интерфейс доступа к API-функциям Winsock. В отличие от интерфейса Winsock, независимо от протокола, элемент управления может использовать лишь IP. Кроме того, он основан на спецификации Winsock 1.1 и ограниченно поддерживает протоколы TCP и UDP.

Сам по себе Winsock не может обратиться к каким-либо параметрам сокетов, что означает недоступность таких функций, как многоадресная и широковещательная рассылка. Обычно элемент управления Winsock полезен, когда необходимы лишь базовые функции работы с сетью — он не обеспечивает оптимальную производительность, поскольку кэширует данные перед их передачей системе и тем самым создает дополнительную нагрузку и неопределенность.

### Свойства

В табл. 15-1 перечислены свойства элемента управления Winsock, которые позволяют управлять поведением и получать информацию о его состоянии.

Табл. 15-1. Свойства элемента управления Winsock

Имя свойства	Тип возвращаемого значения	Доступно только для чтения	Описание
<i>BytesReceived</i>	<i>Long</i>	Да	Возвращает количество байт, ожидающих в буфере приема. Для получения ожидающих данных воспользуйтесь методом <i>GetData</i> .
<i>LocalHostName</i>	<i>String</i>	Да	Возвращает имя локального компьютера.
<i>LocalIP</i>	<i>String</i>	Да	Возвращает строку, содержащую IP-адрес локального компьютера в представлении с точечной нотацией.
<i>LocalPort</i>	<i>Long</i>	Нет	Возвращает или задает используемый локальный порт. Если в качестве номера порта передать 0, система выберет доступный порт произвольно. Обычно так поступают только клиенты.
<i>Protocol</i>	<i>Long</i>	Нет	Возвращает или задает используемый элементом управления протокол, поддерживаются лишь протоколы TCP и UDP. Протоколы обозначаются константами <i>isckTCPProtocolisckUDPProtocol</i> , которым соответствуют значения 0 и 1.
<i>RemoteHost</i>	<i>String</i>	Нет	Возвращает или задает имя удаленного компьютера. Можно использовать как обычное строковое представление имени, так и представление с точечной нотацией.
<i>RemoteHostIP</i>	<i>String</i>	Да	Возвращает IP-адрес удаленного компьютера. Для TCP-подключений значение данного поля задается после успешного установления соединения, для UDP-подключений — при наступлении события <i>DataAmval</i> (поле будет содержать IP-адрес передающего компьютера).
<i>RemotePort</i>	<i>Long</i>	Нет	Возвращает или задает удаленный порт, к которому производится подключение.
<i>SocketHandle</i>	<i>Long</i>	Да	Возвращает значение, соответствующее описателю сокета.
<i>State</i>	<i>Integer</i>	Да	Возвращает состояние элемента управления, являющееся перечислимым типом.

Эти базовые свойства должны быть вам знакомы из материалов главы 7. Они четко соответствуют основным функциям Winsock, используемым в примерах клиент-серверных приложений, обсуждавшихся там. Некоторые свойства, почти не связанные с API Winsock, для корректного использования элемента управления задавать не следует.

Прежде всего необходимо задать свойство *Protocol*, чтобы указать элементу управления требуемый тип сокета — *SOCKJITREAM* или *SOCK\_DGRAM*. Элемент управления действительно создает сокет, и свойство *Protocol* — един-

ственное, что вы можете изменить Свойство *SocketHandle* можно считать после установления соединения или после того, как сервер будет связан и переведен в режим ожидания соединений Это полезно, если требуется передать описатель другим функциям API Winsock, импортированным из DLL-библиотеки

С помощью свойства *State* можно получить информацию о текущем состоянии элемента управления Это очень важно, поскольку элемент управления является асинхронным и события могут происходить в любое время Свойство *State* позволяет гарантировать, что элемент управления находится в требуемом состоянии для последующих действий Перечислим возможные состояния сокета

- III *sckClosed*, значение по умолчанию 0 — сокет закрыт,
  - *sckOpen*, значение 1 — сокет открыт,
- II *sckListening*, значение 2 — сокет прослушивает соединения,
  - *sckConnectionPending*, значение 3 — поступил запрос на соединение, но его обработка еще не завершена,
- III *sckResolvingHost*, значение 4 — идет разрешение имени компьютера,
  - *sckHostResolved*, значение 5 — разрешение имени компьютера завершено,
  - *sckConnecting*, значение 6 — выполнение запроса на соединение начато, но не завершено,
- III *sckConnected*, значение 7 — соединение завершено,
  - *sckClosing*, значение 8 — партнер инициировал закрытие соединения,
  - *sckError*, значение 9 — произошла ошибка

## Методы

Элемент управления Winsock включает лишь несколько методов За некоторыми исключениями, большинство имен методов отражают свои эквиваленты в Winsock Метод для считывания ожидающих данных называется *GetData* Обычно он вызывается при наступлении события *DataArrival*, извещающего о поступлении данных Метод для передачи данных называется *SendData* Метод *PeekData* аналогичен вызову функции Winsock *recv* с параметром *MSGJPEEK* Как обычно, чтение сообщений из памяти отрицательно сказывается на производительности, и поэтому его следует избегать всеми возможными способами Втабл 15-2 перечислены методы элемента управления Winsock и их параметры Сами методы более подробно обсуждаются в других разделах этой главы, посвященных клиент-серверным приложениям

**Табл. 15-2. Методы элемента управления Winsock (не возвращают значений)**

Метод	Параметры	Описание
Accept	RequestID	Только для TCP подключений Используйте метод для приема входящих соединений при обработке события <i>ConnectionRequest</i>

Табл. 15-2. (продолжение)

Метод	Параметры	Описание
<i>Bind</i>	<i>LocalPort</i> <i>LocalIP</i>	Связывает сокет с указанным локальным портом и IP-адресом. Используйте метод при наличии нескольких сетевых адаптеров, вызывая перед методом <i>Listen</i>
<i>Close</i>	Нет	Закрывает соединение или прослушивающий сокет
<i>Connect</i>	<i>RemoteHost</i> <i>RemotePort</i>	Устанавливает TCP-соединение с заданным портом (параметр <i>RemotePort</i> ) удаленного компьютера (параметр <i>RemoteHost</i> )
<i>GetData</i>	<i>Data</i> <i>Type</i> <i>MaxLen</i>	Возвращает ожидающие в настоящий момент данные. Параметры <i>Type</i> и <i>MaxLen</i> являются необязательными: первый определяет тип считываемых данных, второй — указывает количество байт или символов, которое следует считать. Для типов данных, отличных от массива байтов и строки, параметры <i>Type</i> и <i>MaxLen</i> игнорируются.
<i>Listen</i>	Нет	Создает сокет и переводит его в режим прослушивания, используется только для TCP-подключений.
<i>Peek-Data</i>	<i>Data</i> <i>Type</i> <i>MaxLen</i>	Аналогичен методу <i>GetData</i> , но данные не удаляются из системного буфера.
<i>Send-Data</i>	<b>Data</b>	Пересылает данные на удаленный компьютер. Если передана строка в формате UNICODE, она будет предварительно преобразована в формат ANSI. Для передачи двоичных данных всегда используйте тип массив байтов.

## События

События — это асинхронные процедуры, вызываемые в определенной ситуации. Для успешного использования элемента управления Winsock приложение на Visual Basic должно уметь обрабатывать различные события, генерируемые им.

Обычно события наступают в результате действий, предпринимаемых партнером. Например, частичное закрытие TCP происходит, когда одна из сторон TCP-подключения закрывает сокет. Сторона, иницилирующая закрытие, генерирует FIN, а партнер — ACK, чтобы подтвердить запрос на закрытие. На компьютере, принявшем FIN, наступает событие *Close*. Это подсказывает Winsock-приложению, что другая сторона больше не передает данные. Затем приложение считывает оставшиеся данные и вызывает со своей стороны метод *Close*, чтобы полностью закрыть соединение. В табл. 15-3 перечислены все события Winsock.

Табл. 15-3. События элемента управления Winsock

Событие	Аргументы	Наступает
<i>Close</i>	Нет	Когда удаленный компьютер закрывает соединение.
<i>Connect</i>	Нет	После успешного вызова метода <i>Connect</i> .

см след стр

Табл. 15-3.    (продолжение)

Событие	Аргументы	Наступает
Connection-Request	RequestID	Когда удаленный компьютер передает запрос на установление соединения
DataAtrival	bytesTotal	При поступлении новых данных
Error	Number Description Scode Source HelpFile HelpContext CancelDtsplay	При любых ошибках Winsock
SendComplete	Нет	При успешном завершении передачи данных
SendProgress	bytesSent bytesRemaming	При передаче данных

## Пример (UDP-приложения)

Рассмотрим приложение, использующее протокол UDP. Изучите файл проекта Visual Basic с именем SockUDP.vbp, записанный в папке Chapter15 на прилагаемом компакт-диске. После компиляции и запуска проекта на экране появится диалоговое окно, аналогичное приведенному на рис. 15-1. Это приложение лишь принимает и передает UDP-сообщения, и потому для обмена сообщениями можно использовать лишь один его экземпляр.

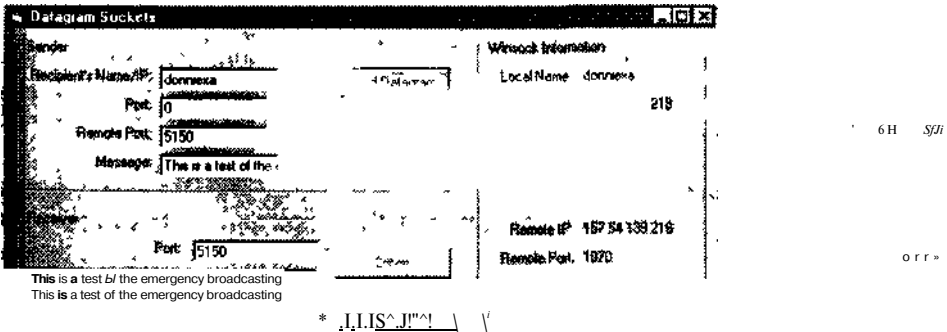


Рис. 15-1. Окно UDP-приложения

На форме имеются два элемента управления Winsock, один из которых принимает, а другой — передает сообщения. Кроме того, присутствуют три рамки групп — для отправителя, для получателя и для общей информации Winsock. В группе получателя следует разместить надписи с именем и IP-адресом принимающего компьютера. Задавая свойство *RemoteHost*, можно использовать либо текстовое имя машины, либо строковое представление IP-адреса с точечной нотацией. При необходимости элемент разрешит имя.

Кроме того, необходим номер удаленного порта, на который вы будете отсылать UDP-пакеты. Обратите также внимание на текстовое поле с номером локального порта — *txtSendLocalPort*.

Для отправителя важен лишь порт, на который вы передаете данные, локальный порт, через который осуществляется передача, значения не имеет. Если оставить номер локального порта равным 0, система назначит неиспользуемый порт.

Последнее текстовое поле — *txtSendData*, предназначено для пересылаемых строковых данных. На форме также размещаются две командные кнопки: одна — для отсылки данных, другая — для закрытия сокета. Для передачи дейтаграмм следует связать элемент Winsock с удаленным адресом, удаленным портом и локальным портом. Это означает, для изменения любого из этих параметров придется сначала закрыть сокет, а затем связать его с новыми параметрами. Именно поэтому на форме присутствует кнопка Close Socket.

В листинге 15-1 приведен код описываемого приложения

# **Листинг 15-1. Пример UDP-приложения**

```
Option Explicit

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdSendOgram_Click()
    ' Если состояние сокета - "закрыт", нам требуется связать сокет
    ' с локальным портом, а также с IP-адресом
    ' и портом удаленного компьютера
    If (sockSend.State = sockClosed) Then
        sockSend.RemoteHost = txtRecipientIP.Text
        sockSend.RemotePort = CInt(txtSendRemotePort.Text)
        sockSend.Bind CInt(txtSendLocalPort.Text)

        cmdCloseSend.Enabled = True
    End If

    ' Теперь мы можем передавать данные

    sockSend.SendData txtSendData.Text
End Sub

Private Sub cmdListen_Click()
    ' Связываем с локальным портом

    sockRecv.Bmd CInt(txtRecvLocalPort.Text)

    ' Отключаем кнопку, поскольку дважды связать сокет с портом
    ' было бы ошибкой (для повторной привязки сокет
    ' необходимо предварительно закрыть)
```

Листинг 15-1. *(продолжение)*

```

cmdListen.Enabled = False
cmdCloseListen.Enabled = True
End Sub

Private Sub cmdCloseSend_Click()
    * Закрываем передающий сокет и отключаем кнопку Close

    sockSend.Close
    cmdCloseSend.Enabled = False
End Sub

Private Sub cmdCloseUsten_Click()
    ' Закрываем прослушивающий сокет

    sockRecv.Close
    ' Включаем кнопки

    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False
    lstRecvData.Clear
End Sub

Private Sub Form_Load()
    ' Инициализируем протоколы сокета, а также
    ' задаем некоторые значения и надписи по умолчанию

    sockSend.Protocol = sckUDPProtocol
    sockRecv.Protocol = sckUDPProtocol

    lblHostName.Caption = sockSend.LocalHostName
    lblLocalIP.Caption = sockSend.LocalIP

    cmdCloseListen.Enabled = False
    cmdCloseSend.Enabled = False

    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub

Private Sub sockSend_Error(ByVal Number As Integer, _
    Description As String, ByVal Scode As Long,
    ByVal Source As String, ByVal HelpFile As String, _
    ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub sockRecv_DataArrival(ByVal bytesTotal As Long)
    Dim data As String

    ' Выделяем строку достаточного размера и получаем данные;

```

## Листинг 15-1. (продолжение)

Hi

```

' затем добавляем их в описок
data = String(bytesTotal + 2, Chr$(0))
sockRecv.GetData data, , bytesTotal
lstRecvData.AddItem data
' Обновляем надписи с удаленным IP-адресом и портом

lblRemoteIP.Caption = sockRecv.RemoteHostIP
lblRemotePort.Caption = sockRecv.RemotePort
End Sub

Private Sub sockRecv_Error(ByVal Number As Integer,
    Description As String, ByVal Scode As Long,
    ByVal Source As String, ByVal HelpFile As String,
    ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub Timer1_Timer()
    ' Когда срабатывает таймер, надписи обновляются
    ' информацией о состоянии сокета

    Select Case sockSend.State
        Case sckClosed
            lblSenderState.Caption = 'sckClosed"
        Case sckOpen
            lblSenderState.Caption = 'sckOpen"
        Case sckListening
            lblSenderState.Caption = 'sckListening"
        Case sckConnectionPending
            lblSenderState.Caption = 'sckConnectionPending"
        Case sckResolvingHost
            lblSenderState.Caption = 'sckResolvingHost"
        Case sckHostResolved
            lblSenderState.Caption = 'sckHostResolved"
        Case sckConnecting
            lblSenderState.Caption = 'sckConnecting"
        Case sckClosing
            lblSenderState.Caption = 'sckClosing"
        Case sckError
            lblSenderState.Caption = 'sckError"
        Case Else
            lblSenderState.Caption = "unknown"
    End Select
    Select Case sockRecv.State
        Case sckClosed
            lblReceiverState.Caption = "sckClosed"
        Case sckOpen
            lblReceiverState.Caption = "sckOpen"
    End Select

```

см. след. стр.



## Листинг 15-1.    {продолжение)

```

Case sckListening
    lblReceiverState.Caption = "sckListening"
Case sckConnectlonPending
    lblReceiverState.Caption = "sckConnectionPending"
Case sckResolvingHost
    lblReceiverState.Caption = "sckResolvingHost"
Case sckHostResolved
    lblReceiverState.Caption = "sckHostResolved"
Case sckConnecting
    lblReceiverState.Caption = "sckConnecting"
Case sckClosing
    lblReceiverState.Caption = "sckClosing"
Case sckError
    lblReceiverState.Caption = "sckError"
Case Else
    lblReceiverState.Caption = "unknown"
End Select
End Sub

```

## Пересылка UDP-сообщений

Теперь изучим код программы обмена сообщениями. Обратите внимание на процедуру *Form\_Load*. Сначала требуется задать свойству *Protocol* элемента управления Winsock *sockSend* значение *sckUDPProtocol*. Другие команды в данной процедуре не влияют на функциональность пересылки. Кнопку *and-CloseSend* следует отключить для завершенности, поскольку при вызове метода *Close* для уже отключенного элемента управления ничего не происходит. Заметьте, что элемент управления Winsock по умолчанию закрыт.

Далее мы рассмотрим процедуру *cmdSendDgram\_Click*, которая вызывает-ся щелчком кнопки Send Data. Это основа механизма, отсылающего UDP-сообщения. Прежде всего проверяется состояние сокета. Если сокет закрыт, его связывают с удаленным адресом, а также с локальным и удаленным портами — состояние элемента управления изменится с *sckClosed* на *sckOpen*. Если код не выполнит эту проверку и будет связывать сокет при каждой передаче данных, произойдет генерация ошибки 40020: Invalid operation at current state. Сокет остается связанным с заданными параметрами до своего закрытия. Именно поэтому код активирует кнопку Close Socket для передающего сокета после привязки элемента управления.

Последний этап — вызов метода *SendData* и передача ему данных, которые требуется переслать. Когда метод *SendData* возвращается, это означает, что код завершил пересылку данных.

С пересылкой UDP-сообщений связаны две другие подпроцедуры. Первая называется *cmdCloseSend* — она, как и следует из названия, закрывает передающий сокет, позволяя пользователю перед повторной передачей данных изменить имя удаленного компьютера, а также номер удаленного или локального порта. Вторая подпроцедура называется *sockSend\_Error* и является событием Winsock, которое наступает при любой ошибке Winsock. Поскольку протокол

UDP не надежен, будет генерироваться немного ошибок. В случае ошибки код просто выдает ее описание. Единственное сообщение, выводимое данным приложением, — о недоступности получателя.

## Прием UDP-сообщений

Как видите, передавать UDP-пакеты с использованием элемента управления Winsock просто и удобно. Принимать UDP-пакеты еще легче. Вернемся к процедуре *FormLoad*. Как и в случае передающего элемента управления Winsock, код задает свойству *Protocol* значение UDP, а также отключает кнопку *Close Listen*. Заккрытие уже закрытого сокета не создает каких-либо проблем, однако код все же отключает кнопку для завершенности. Кроме того, на разных стадиях проектирования приложения вам следует задаваться вопросом: «Что произойдет, если я вызову метод X?». Источник большинства проблем разработчиков при использовании элемента управления Winsock — вызов метода при неверном состоянии сокета (например, вызов метода *Connect* для уже установившего соединение элемента управления Winsock).

Рассмотрим процедуру *cmdListen Click*, которая осуществляет прослушивание входящих UDP-пакетов. Это обработчик кнопки *Listen*. Единственное необходимое действие — вызвать метод *Bind* на принимающем элементе управления Winsock и передать номер локального порта, на котором пользователь хочет прослушивать входящие UDP-дейтаграммы. При прослушивании входящих UDP-пакетов коду требуется лишь номер локального порта — номер удаленного порта, на который пересылаются данные, значения не имеет. После того как код свяжет элемент управления, он отключит кнопку *cmdListen*, тем самым предотвращая возможность повторного щелчка кнопки *Listen* пользователем (при попытке связать уже связанный элемент управления произойдет ошибка времени выполнения).

На этом этапе элемент управления *sockRecv* регистрируется для получения UDP-данных. Когда на порт, с которым он связан, приходят UDP-данные, наступает событие *DataArrival*, реализованное в процедуре *sockRecv JDataArrival*. Значение параметра *bytesTotal*, который передается событию — число доступных для считывания байт. Код выделяет строку чуть большего размера, чем объем считываемых данных. Затем вызывается метод *GetData* и выделенная строка передается в качестве первого параметра. По умолчанию тип второго параметра — *vbString*. Третий параметр указывает число байт, которые требуется считать, в нашем примере это число определяется значением параметра *bytesTotal*. Если код передает запрос на считывание меньшего числа байт, чем значение параметра, генерируется ошибка времени выполнения.

После того как данные считаны и помещены в буфер символов, код добавляет их в список считанных сообщений. Затем подпроцедура задает заголовки меток, отображающих IP-адрес и номер порта удаленного компьютера. При получении любого UDP-пакета свойствам *RemoteHostIP* и *RemotePort* задаются значения, соответствующие IP-адресу и номеру порта удаленного компьютера, через который был передан принятый пакет. Таким образом, если приложение получает UDP-пакеты от нескольких компьютеров, значения этих свойств будут часто изменяться.

Последние две подпроцедуры, связанные с приемом UDP-сообщений — *cmdCloseListen\_Click* и *sockRecv\_Error*. Обработчик *cmdCloseListen\_Click* срабатывает при щелчке кнопки Close Listen. Процедура лишь вызывает метод *Close* для элемента управления Winsock. При закрытии UDP-элемента управления освобождается базовый описатель сокета. Событие *sockRecv\_Error* вызывается при любой ошибке Winsock. Как уже упоминалось, из-за неустойчивой природы будет генерироваться мало ошибок UDP.

## Получение информации от элемента Winsock

Последняя часть рассматриваемого примера — рамка группы Winsock Information. Заголовки меток, содержащих локальное имя и локальный IP-адрес задаются в момент загрузки формы. После загрузки формы и создания экземпляров элементов управления Winsock свойствам *LocalHostName* и *LocalIP* назначают значения, соответствующие имени и IP-адресу компьютера. Эти значения можно считать в любое время.

Следующие две надписи — Sender State и Receiver State, отображают состояние двух используемых приложением элементов управления Winsock. Сведения о состоянии обновляются два раза в секунду. Здесь в работу включается элемент управления Timer: каждые 500 миллисекунд он вызывает обработчик Timer, который запрашивает о состоянии сокетов и обновляет надписи. Мы выводим сведения о состоянии сокетов для информации. Значения последних двух меток — Remote IP и Remote Port, задают при получении UDP-сообщения.

## Запуск UDP-приложения

Теперь рассмотрим UDP-приложение в работе. Лучший способ протестировать его — запустить экземпляр приложения на трех отдельных компьютерах. В одном из приложений щелкните кнопку Listen, а в двух других — введите в поле Recipient's Name/IP имя или IP-адрес компьютера, на котором выполняется первый экземпляр. Теперь несколько раз щелкните кнопку Send Datagram — в соответствующем окне принимающего приложения должны отобразиться сообщения. При приеме каждого сообщения поля группы Winsock Information обновляются и им задаются значения, соответствующие IP-адресу отправителя и номеру порта, через который было передано сообщение. Вы можете воспользоваться командами элемента управления Sender этого же экземпляра приложения, чтобы принимать сообщения на том же компьютере.

Еще один интересный тест — широковещательная рассылка по определенной подсети или рассылка широковещательных дейтаграмм. После того как вы отошлете дейтаграмму в определенную подсеть, сообщение получат все прослушивающие приложения (при условии, что все три компьютера тестируются в одной подсети).

Рассмотрим пример: среди тестовых компьютеров имеется две системы с одним сетевым адаптером, их IP-адреса — 157.54.185.186 и 157.54.185.224. На третьем компьютере установлено несколько сетевых адаптеров; IP-адрес-

са этой системы — 169.254.26.113 и 157.54.185-206. Как видите, все три системы принадлежат к одной подсети — 157.54.185.255.

Давайте на секунду отвлечемся и обсудим одну важную деталь. Для получения UDP-сообщений требуется при вызове метода *Bind* явно связать элемент управления с первым IP-адресом, хранящимся в сетевых привязках. Этого достаточно, если в системе установлен лишь один сетевой адаптер. Ну, а если на компьютере несколько сетевых интерфейсов, и следовательно, несколько IP-адресов? В таком случае второй параметр, передаваемый методу *Bind*, — IP-адрес, с которым связан элемент управления Winsock. К сожалению, свойство элемента управления Winsock возвращает лишь один IP-адрес, методов, позволяющих получить остальные IP-адреса локального компьютера, не существует.

Теперь попробуем осуществить широковещательную рассылку. Закройте все передающие и прослушивающие сокеты во всех запущенных экземплярах приложения. На компьютере с одним сетевым адаптером щелкните кнопку Listen, чтобы каждая система могла получать сообщения дейтаграмм. Компьютер с несколькими сетевыми адаптерами не используется, поскольку в коде элементы управления не связываются с каким-либо конкретным IP-адресом. На третьем компьютере введите адрес получателя — 157.54.185.255, и несколько раз щелкните кнопку Send Data. Вы увидите, что сообщение принимают оба приложения.

Вам, возможно, будет интересно, как система выбирает интерфейс для передачи дейтаграммы, если на передающем компьютере установлено несколько сетевых адаптеров. Одна из функций таблицы маршрутизации — определить оптимальный интерфейс для передачи сообщения при наличии конечного адреса и адрес сетевых адаптеров, установленных на локальном компьютере<sup>1</sup>.

Последний тест таков: закройте передающий сокет на третьем компьютере, задайте адрес получателя как 255.255.255.255 и несколько раз щелкните кнопку Send Datagram. Результаты должны быть одинаковыми: два других прослушивающих приложения получают сообщение. На компьютере с несколькими сетевыми адаптерами будет наблюдаться единственное отличие — UDP-сообщение рассылается по всем сетям, с которыми он соединен.

## Состояние UDP-сокетов

Вас, возможно, запутал порядок, в котором должен осуществляться вызов методов для успешного приема-передачи дейтаграмм. Как упоминалось ранее, самая распространенная ошибка при программировании элемента управления Winsock — вызов метода, работа которого при текущем состоянии сокета невозможна. Чтобы избежать таких ошибок, изучите диаграмму на рис. 15-2. Она иллюстрирует состояния сокетов при передаче UDP-сообщения.

<sup>1</sup>Подробности о подсетях и маршрутизации см. в литературе, посвященной пакету протоколов TCP/IP, например, в книге Ричарда Стивенса «TCP/IP Illustrated Volume I» (Addison-Wesley, 1994), или доктора Сидни Фейт «TCP/IP Architecture, Protocols, and Implementation with IP v6 and IP Security» (McGraw-Hill, 1996).

ний. Обратите внимание, что начальное состояние по умолчанию — *sckClosed*, и при указании неверного имени компьютера не генерируется ошибок.

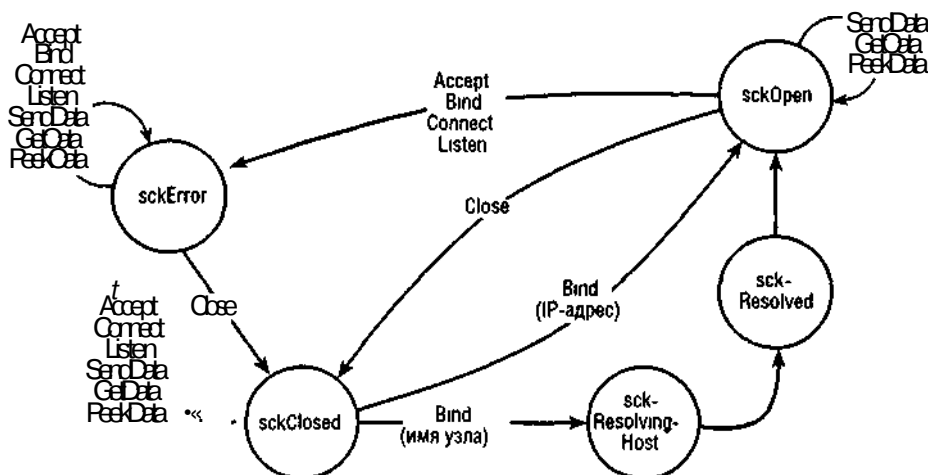


Рис. 15-2. Диаграмма состояний UDP-сокета

## Пример (TCP-приложение)

Использование элемента управления Winsock с протоколом TCP несколько сложнее, но более распространено, чем с протоколом UDP. Как и в случае UDP, мы рассмотрим пример TCP-приложения и подробно изучим его код, чтобы понять этапы, необходимые для успешного установления TCP-соединения. На рис. 15-3 изображено запущенное приложение.

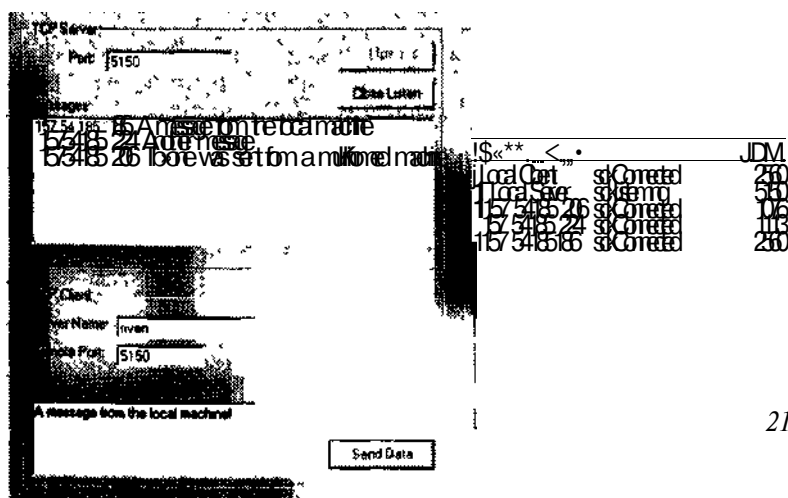


Рис. 15-3. Пример TCP-приложения

\* В форме на рис. 15-3 три группы флажков: TCP Server, TCP Client и Winsock Information. Сначала рассмотрим область TCP Server. У сервера есть текстовое поле — *txtServerPort*, для локального порта, к которому сервер будет привязан для прослушивания входящих соединений клиентов. Кроме того на форме имеются две кнопки: одна — для перевода сервера в режим прослушивания и вторая — для остановки и прекращения приема входящих соединений. Наконец, на рисунке показан единственный элемент Winsock — *sockServer*. Если вы взглянете на страницу его свойств, то увидите, что свойство *Index* равно 0. Это значит, что данный элемент фактически является массивом, где может содержаться несколько экземпляров элемента Winsock. Значение 0 указывает, что во время загрузки формы будет создан только один экземпляр (нулевой элемент массива). В любое время можно динамически загрузить еще один экземпляр элемента Winsock в массив этого элемента.

Массив элементов Winsock — основа серверных возможностей. Помните, что с одним элементом Winsock связан лишь один описатель сокета. В главе 7 упоминалось, что когда сервер принимает входящее соединение, для работы с этим соединением создается новый сокет. Наше приложение сконструировано для динамической загрузки элементов Winsock в ответ на запрос о соединении от клиента, поэтому соединение можно передать вновь загруженному элементу, не прерывая его обработки сокетом на сервере.

Другой способ — создать *x* элементов Winsock еще на этапе проектирования. Впрочем, это решение требует значительных ресурсов и плохо масштабируется. При старте приложения потребуется много времени, чтобы загрузить все необходимые ресурсы для каждого элемента; кроме того, не всегда ясно, сколько элементов необходимо. Разместив *x* элементов, вы жестко ограничите количество одновременно обслуживаемых клиентов. Если требования вашего приложения допускают фиксирование количества одновременных соединений, проще реализовать вторую модель, а не массив. Впрочем, для большинства приложений массив элементов Winsock подойдет лучше.

В листинге 15-2 приведен код программы. Файл с кодом данного проекта Visual Basic — *SocketTCP.vbp*, записан в папке Chapter15 на прилагаемом компакт-диске.

»1

## Листинг 15-2. Пример TCP-приложения

Option Explicit

<sup>1</sup> Индексное значение последнего элемента **управления**

<sup>1</sup> Winsock автоматически загружается в массив **sockServer**

Private ServerIndex As Long

Private Sub cmdCloseListen\_Click()

Dim itemX As Object

- ' Закрываем прослушивающий сокет сервера. После этого клиенты
- не смогут подключаться к серверу

см. след. стр.

ЛИСТИНГ 15-2. (продолжение)

```

sockServer(0).Close
cmdListen.Enabled = True
cmdCloseListen.Enabled = False

Set itemx = lstStates.ListItems.Item(2)
itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdConnect_Click()
    ' Заставляем клиентский элемент управления попытаться
    ' подключиться к заданному серверу
    ' по указанному номеру порта

    sockClient.LocalPort = 0
    sockClient.RemoteHost = txtServerName.Text
    sockClient.RemotePort = CInt(txtPort.Text)
    sockClient.Connect

    cmdConnect.Enabled = False
End Sub

Private Sub cmdDisconnect_Click()
    Dim itemx As Object
    ' Закрываем клиентское подключение и настраиваем
    ' командные кнопки для последующих соединений

    sockClient.Close

    cmdConnect.Enabled = True
    cmdSendData.Enabled = False
    cmdDisconnect.Enabled = False
    ' Задаем номер порта как -1, чтобы указать отсутствие соединения

    Set itemx = lstStates.ListItems.Item(i)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdListen_Click()
    Dim itemx As Object
    ' Переводим серверный элемент управления в режим прослушивания
    ' на порту с заданным номером

    sockServer(0).LocalPort = CInt(txtServerPort.Text)
    sockServer(0).Listen

    Set itemx = lstStates.ListItems.Item(2)

```

Листинг 15-2. (продолжение)

```

itemx.SubItems(2) = sockServer(0).LocalPort      Te\ .f

cmdCloseListen.Enabled = True
cmdListen.Enabled = False
End Sub

Private Sub cmdSendData_Click()
    ' При наличии соединения передаем указанные данные на сервер      uut

    If (sockClient.State = sckConnected) Then
        sockClient.SendData txtSendData.Text      Q f A
    Else
        MsgBox "Unexpected error! Connection closed"
        Call cmdDisconnect_Click
    End If
End Sub

Private Sub Form_Load()
    ' УЛ ' *r,ils. лей
    Dim itemx As Object      duS ЫлЗ

    lblLocalHostname.Caption = sockServer(0).LocalHostName
    lblLocalHostIP.Caption = sockServer(0).LocalIP

    ' Инициализируем свойство Protocol со значением sckTCPProtocol,
    ' мы будем использовать только протокол TCP      ' Yti

    ServerIndex = 0
    sockServer(0).Protocol = sckTCPProtocol
    sockClient.Protocol = sckTCPProtocol      даясмоса
    ' Настраиваем кнопки

    cmdDisconnect.Enabled = False      " W I T * «„а»n.      duS
    cmdSendData.Enabled = False      *?
    cmdCloseListen.Enabled = False
    ' Инициализируем элемент управления ListView, содержащий
    ' информацию о текущем состоянии всех созданных элементов
    ' управления Winsock (необязательно подключенных или используемых)

    Set itemx = lstStates.ListItems.Add(1, , "Local Client")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    Set itemx = lstStates.ListItems.Add(2, , "Local Server")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    ' Инициализируем таймер, который управляет скоростью
    ' обновления информации об упоминавшихся выше состояниях сокетов

```



Листинг 15-2.    (продолжение)

```

TimeM.Interval = 500
Timeri.Enabled = True
End Sub

```

```

Private Sub sockClient_Close()
    sockClient.Close
End Sub

```

```

Private Sub sockClient_Connect()
    Dim itemx As Object

```

```

' Соединение было успешным, активируем кнопки передачи данных

```

```

cmdSendData.Enabled = True

```

```

cmdDisconnect.Enabled = True

```

```

Set itemx = lstStates.ListItems.Item(i)

```

```

itemx.SubItems(2) = sockClient.LocalPort

```

```

End Sub

```

```

Private Sub sockClient_Error(ByVal Number As Integer,
    Description As String, ByVal Scode As Long, _
    ByVal Source As String, ByVal HelpFile As String, _
    ByVal HelpContext As Long, CancelDisplay As Boolean)

```

```

' У элемента управления Client произошла ошибка: выводим

```

```

' сообщение и закрываем элемент управления. Ошибка переводит

```

```

' элемент управления в состояние sckError; переход в другое состояние

```

```

' возможен после вызова метода Close.

```

```

MsgBox Description

```

```

sockClient.Close

```

```

cmdConnect.Enabled = True

```

```

End Sub

```

```

Private Sub sockServer_Close(index As Integer)

```

```

    Dim itemx As Object

```

```

    ' Закрываем данный элемент управления Winsock

```

```

    sockServer(index).Close

```

```

    Set itemx = lstStates.ListItems.Item(index + 2)

```

```

    lstStates.ListItems.Item(index + 2).Text = "-.-.-"

```

```

    itemx.SubItems(2) = "-1"

```

```

End Sub

```

```

Private Sub sockServer_ConnectionRequest(Index As Integer, _
    ByVal requestID As Long)

```

```

    Dim i As Long, place As Long, freeSock As Long, itemx As Object

```

## Листинг 15-2. {продолжение}

```

' Просматриваем массив и ищем закрытый элемент управления,
' который можно повторно использовать
freeSock = 0
For i = 1 To ServerIndex
    If sockServer(i).State = sckClosed Then
        freeSock = 1
        Exit For
    End If
Next i
' Если значение freeSock по-прежнему равно 0, свободные элементы [
' управления отсутствуют, и поэтому мы загружаем
' новый элемент управления Winsock
If freeSock = 0 Then
    ServerIndex = ServerIndex + 1
    Load sockServer(ServerIndex)
    sockServer(ServerIndex).Accept requestID
    place = ServerIndex
Else
    sockServer(freeSock).Accept requestID
    place = freeSock
End If
' Если не было обнаружено свободных элементов управления,
' выше мы добавляем дополнительный элемент. Создайте
' в элементе управления ListView запись для нового элемента
' управления. В любом случае установите состояние нового
' соединения как sckConnected.
If freeSock = 0 Then
    Set itemx = lstStates.ListItems.Add(, ,
        sockServer(ServerIndex).RemoteHostIP)
Else
    Set itemx = lstStates.ListItems.Item(freeSock + 2)
    lstStates.ListItems.Item(freeSock + 2).Text = _
        sockServer(freeSock).RemoteHostIP
End If
itemx.SubItems(2) = sockServer(place).RemotePort

End Sub

Private Sub sockServer_DataArrival(mdex As Integer,
    ByVal bytesTotal As Long)
    Dim data As String, entry As String

    ' Выделяем строковый буфер достаточного размера и получаем данные

```

см след стр

## Листинг 15-2. {продолжение}

&lt;\*

```

data = String(bytesTotal + 2, Chr$(0))
sockServer(index).GetData data, vbString, bytesTotal
' Добавляем IP-адрес клиента в начало сообщения
' и помещаем сообщение в список
entry = sockServer(index).RemoteHostIP & ": " &
lstMessages.AddItem entry
End Sub

Private Sub sockServer_Error(index As Integer, _
    ByVal Number As Integer, Description As String, _
    ByVal Scode As Long, ByVal Source As String, _
    ByVal HelpFile As String, ByVal HelpContext As Long, _
    CancelDisplay As Boolean)
' Выводим сообщение об ошибке и закрываем указанный элемент
' управления. Ошибка переводит элемент управления в состояние
' sckError, которое снимается лишь после вызова метода Close.
MsgBox Description
sockServer(index).Close
End Sub

Private Sub Timer1_Timer()
Dim i As Long, index As Long, itemx As Object
' Задаем состояние локального элемента управления Winsock клиента
Set itemx = lstStates.ListItems.Item(i)
Select Case sockClient.State
Case sckClosed
    itemx.SubItems(i) = "sckClosed"
Case sckOpen
    itemx.SubItems(i) = "sckOpen"
Case sckListening
    itemx.SubItems(1) = "sckListening"
Case sckConnectionPending
    itemx.SubItems(1) = "sckConnectionPending"
Case sckResolvingHost
    itemx.SubItems(1) = "sckResolvingHost"
Case sckHostResolved
    itemx.SubItems(i) = "sckHostResolved"
Case sckConnecting
    itemx.SubItems(1) = "sckConnecting"
Case sckConnected
    itemx.SubItems(i) = "sckConnected"
Case sckClosing
    itemx.SubItems(1) = "sckClosing"
Case sckError
    itemx.SubItems(i) = "sckError"

```

Листинг 15-2. (продолжение)

```

Case Else
    itemx.SubItems(i) = "unknown: " & sockClient.State
End Select
    Теперь задаем состояния прослушивающего серверного элемента
    ' управления, а также элементов управления всех подсоединенных
1  клиентов

index = 0
For i = 2 To ServerIndex + 2
    Set itemx = lstStates.ListItems.Item(i)
*И
Г
    Select Case sockServer(index).State
        Case sckClosed
            itemx.SubItems(i) = "sckClosed"
        Case sckOpen
            itemx.SubItems(i) = "sckOpen"
        Case sckListening
            itemx.SubItems(i) = "sckListening"
        Case sckConnectionPending
            itemx.SubItemsd) = "sckConnectionPending"
        Case sckResolvingHost
            itemx.SubItems(1) = "sckResolvingHost"
        Case sckHostResolved
            itemx.SubItemsd) = "sckHostResolved"
        Case sckConnecting
            itemx.SubItems(1) = "sckConnecting"
        Case sckConnected
            itemx.SubItems(1) = "sckConnected"
        Case sckClosing
            itemx.SubItems(1) = "sckClosing"
        Case sckError
            itemx.SubItemsd) = "sckError"
        Case Else
            itemx.SubItemsd) = "unknown"
    End Select
    index = index + 1
Next i
End Sub

```

## ТСР-сервер

Рассмотрим код формы. Обратите внимание на процедуру *Form Load* из листинга 15-2. Два первых параметра лишь присваивают надписям значения, соответствующие имени и IP-адресу локального компьютера. Эти надписи находятся в рамке группы Winsock Information, которая служит тем же целям, что и информационная группа из примера с UDP. Далее серверный элемент управления *sockServer* инициализируется с параметром *sckTCPProtocol*. Нулевой элемент массива элементов управления Winsock всегда является

прослушивающим сокетом. Затем процедура отключает кнопку Close Listen, которая активируется, лишь когда сервер снова начнет прослушивать клиентские соединения.

Последняя часть процедуры настраивает элемент управления *ListView* с именем *lstStates*. Он отображает состояние каждого используемого в настоящий момент элемента управления Winsock. Код добавляет записи для клиентского и серверного элемента управления, и они становятся элементами 1 и 2, соответственно. Все прочие динамически загружаемые элементы управления Winsock добавляются после этих двух записей. Имя записи о серверном элементе управления — Local Server. Как и в примере с протоколом UDP, процедура настраивает таймер для управления частотой обновления состояний сокетов. По умолчанию триггеры таймеров обновляются каждые полсекунды.

Теперь давайте рассмотрим две кнопки, используемые сервером. Первая — Listen, инициирует несложное действие. Обработчик этой кнопки — *cmdListen*, задает свойству значение, введенное пользователем в поле *txtServerPort*. Поле с номером локального порта наиболее важно для прослушивающего сокета. Именно к порту с этим номером клиенты пытаются подключиться, чтобы установить соединение. Задав свойство *LocalPort*, код вызывает метод *Listen*. Как только обработчик кнопки Listen переведет элемент управления *sockServer* в прослушивающее состояние, программа начнет ожидать, когда для элемента управления *sockServer* наступит событие *ConnectionRequest*, указывающее, что клиент пытается установить соединение. Кнопка Close Listen закрывает элемент управления *sockServer*. Обработчик кнопки Close Listen вызывает метод *Close* для элемента *sockServer(0)*, запрещая установление соединений клиентами.

Наиболее важное событие для TCP-сервера — *ConnectionRequest*, оно обрабатывает входящие запросы клиентов. Поступивший от клиента запрос на соединение можно обработать двумя способами. Первый — воспользоваться непосредственно сокетом сервера. Вызовите метод *Accept* для элемента управления сервера с параметром *requestID*, который передается обработчику события — это пример действий SockTCP. Недостаток данного метода — прослушивающий сокет будет закрыт и другие клиенты не смогут установить соединение с сервером.

Помните, что нулевой элемент массива элементов управления Winsock является прослушивающим сокетом. Прежде всего просмотрите массив и найдите элемент управления с состоянием «закрыт» (например, в запросе укажите, что значение свойства *State* должно равняться *sockClosed*). Конечно, на первом этапе вы не обнаружите свободных элементов управления, поскольку они вообще не загружены. В этом случае первый цикл не выполняется и значение переменной *freeSock* будет по-прежнему равно 0, сообщая, что свободные элементы управления не обнаружены. Следующие этапы динамически загружают новый элемент управления Winsock, увеличивая значение счетчика *ServerIndex* (место в массиве, куда следует загрузить элемент управления), и затем выполняют следующий оператор

```
Load sockServer(ServerIndex)
```

Теперь, когда новый элемент Winsock загружен, процедура может вызвать метод *Accept* с указанным идентификатором запроса (request ID). Оставшиеся операторы добавляют новую запись в элемент управления ListView с именем *IstStates*, чтобы приложение могло отображать текущее состояние нового элемента управления Winsock.

Если состояние уже загруженного элемента Winsock — «закрыт», процедура повторно воспользуется данным элементом управления, вызвав для него метод *Accept*. Непрерывная загрузка и выгрузка элементов управления занимает много процессорного времени и снижает производительность. Кроме того, при выгрузке элемента управления Winsock может возникнуть утечка памяти (подробней — далее в этой главе).

Оставшиеся серверные функции просты. Событие *sockServer\_Close* наступает, когда клиент вызывает метод *Close*. Тогда сервер закрывает со своей стороны сокет и обнуляет запись об IP-адресе в элементе управления ListView, присваивая ей значение «—•» и задавая номер порта как -1. Функция *sockServer\_DataAmval* выделяет буфер для приема данных, а затем вызывает метод *GetData* для считывания информации. В результате в список *IstMessages* добавляется сообщение. Последняя серверная функция — обработчик события *Error*. При ошибке обработчик выводит соответствующее сообщение и закрывает элемент управления.

## TCP-клиент

Теперь рассмотрим код клиентской части. Единственная инициализация, выполняемая клиентом в процедуре *Form\_Load*, — выбор протокола TCP для элемента управления *sockdient*. Не входящие в код инициализации три обработчика командных кнопок связаны с клиентом и некоторыми обработчиками событий. Первая кнопка — Connect, ее обработчику — *cmdConnect\_ChkLocalPort*, задано значение 0, поскольку номер порта, выбранного на локальном компьютере, не важен. Параметры *RemoteHost* и *RemotePort* задаются в соответствии со значениями полей *txtServerName* и *txtPort*. Это вся информация, необходимая для установки TCP-соединения с сервером.

Единственное, что осталось сделать — вызвать метод *Connect*. После этого элемент управления Winsock будет находиться в процессе разрешения имени или установки соединения (состояние элемента управления — *sckResolvingHost*, *sckResolved* или *sckConnecting*). Когда соединение установлено, состояние изменится на *sckConnected* и наступит событие *Connect*. Далее рассматриваются различные состояния элемента управления и переходы между ними.

После установления соединения вызывается обработчик *sockClient\_Connect*, который активирует кнопки Send Data и Disconnect. Кроме того, для записи Local Client в элементе управления ListView с именем *IstStates* будет обновлен номер порта локального компьютера, через который устанавливается соединение. Теперь вы можете принимать и передавать данные.

Существуют и два других обработчика событий *sockClient\_Close* и *sockClient\_Error*. Обработчик события *sockClient\_Close* лишь закрывает элемент управления Winsock, а обработчик события *sockClient\_Error* — выводит окно сообщения с описанием ошибки и затем закрывает элемент управления.

Две оставшихся части кода клиента — командные кнопки *Send Data* и *Disconnect*. Кнопку *Send Data* обрабатывает подпроцедура *cmdSendData\_Click*. Если состояние элемента управления Winsock — «подключен», процедура вызывает метод *SendData*, передавая ему строку из поля *txtSendData*. Кнопку *Disconnect* обрабатывает подпроцедура *cmdDisconnect\_Click*. Данный обработчик лишь закрывает клиентский элемент управления, возвращает некоторые кнопки в исходное состояние, а также обновляет запись *Local Client* в элементе управления *lstStates*.

## Получение информации о состоянии элемента управления Winsock

Последняя часть примера, использующего протокол TCP, — рамка группы Winsock Information. Мы уже обсуждали подобную рамку, однако для ясности вкратце рассмотрим ее повторно.

Как и в примере с UDP, таймер управляет обновлением информации о текущем состоянии всех загруженных элементов управления Winsock. Периодичность обновления по умолчанию — 500 миллисекунд. При загрузке в элемент управления Listview с именем *IstStates* добавляются две записи. Первая — надпись *Local Client*, соответствующая клиентскому элементу управления Winsock *sockClient*. Вторая — *Local Server*, относится к прослушивающему сокету. При установлении нового клиентского соединения динамически загружается новый элемент управления Winsock и в элемент управления *sockStates* добавляется новая запись — IP-адрес клиента. После отключения клиента запись возвращается в исходное состояние — IP-адрес «—.—.—», номер порта — (-1). Конечно, подключающийся новый клиент повторно использует неиспользуемые элементы из серверного массива. Так же отображаются IP-адрес и имя локального компьютера.

## Запуск TCP-приложения

Запуск TCP-приложения не вызывает каких-либо трудностей. Запустите по экземпляру приложения на трех отдельных компьютерах. При использовании TCP не имеет значения, сколько сетевых адаптеров установлено на компьютерах, поскольку оптимальный интерфейс для данного TCP-соединения выбирает таблица маршрутизации. В одном из приложений щелкните кнопку *Listen*. Вы увидите, что значение записи *Local Server* в элементе управления ListView с именем *State Information* изменилось с *sockClosed* на *sockListening*, а номер порта задан как 5150. Теперь сервер может принимать клиентские запросы на соединение.

На одном из клиентов задайте полю *Server Name* значение, соответствующее имени компьютера, на котором выполняется первый экземпляр приложения (прослушивающий сервер) и затем щелкните кнопку *Connect*. Элемент управления, которому в клиентском приложении соответствует запись *Local Client* из списка *State Information*, теперь находится в состоянии *sockConnected*, а в качестве номера порта, через который установлено соединение, отображается неотрицательное число. Кроме того, на сервере в список

State Information добавляется запись — IP-адрес только что подключившегося клиента. Состояние новой записи — *sockConnected*, она также содержит номер порта сервера, с которым установлено соединение.

Теперь введите текст в поле Message клиентского приложения и несколько раз щелкните кнопку Send Data. Вы увидите сообщения, появляющиеся в окне списка Messages серверного приложения. После этого отключите клиент, щелкнув кнопку Disconnect. На клиентском компьютере записи Local Client из списка State Information будет назначено состояние *sockClosed*, а номер порта задан как — 1. На сервере запись, соответствующая клиенту, не удаляется, а лишь помечается как неиспользуемая. Ее имя задается, как IP-адрес «—.—.—.—», номер порта — как -1 и состояние — как *sockClosed*.

На третьем компьютере введите в поле Server Name имя прослушивающего сервера и установите клиентское подключение. Вы получите те же результаты, что и в случае с первым клиентом, но для обработки второго клиента сервер будет использовать тот же элемент управления Winsock. Если состояние элемента Winsock — «закрыт», он может применяться для принятия любых входящих соединений. Последнее, что мы советуем сделать, — используйте клиент на стороне сервера, чтобы установить соединение локально. После того, как соединение будет установлено, в список Socket Information добавится новая запись. Единственное отличие в том, что теперь указан IP-адрес, соответствующий IP-адресу сервера. Поработайте с клиентскими и серверными приложениями, чтобы понять, как они взаимодействуют и какие результаты дает выполнение каждой из команд.

## Состояние TCP-сокетов

Использовать элемент управления Winsock с протоколом TCP значительно сложнее, чем работать с UDP-сокетами, поскольку здесь гораздо больше состояний сокетов. На рис. 15-4 приведена диаграмма состояний TCP-сокета. Начальное состояние по умолчанию — *sockClosed*. Переходы между состояниями просты и не требуют каких-либо комментариев, за исключением состояния *sockClosing*.

Из-за частичного закрытия TCP метод *SendData* может перейти из этого состояния в другое двумя способами. Как только одна из сторон TCP-соединения вызовет метод *Close*, она больше не сможет передавать данные. Другая сторона соединения получает событие *Close* и переходит в состояние *sockClosing*, но по-прежнему может передавать информацию. Именно поэтому для метода *SendData* существует два пути перехода из состояния *sockClosing*. Если сторона, вызвавшая метод *Close*, пытается вызвать *SendData*, генерируется ошибка и состояние элемента управления Winsock изменяется на *sockError*. Сторона, получившая событие *Close*, может свободно передавать информацию и принять все оставшиеся данные.

## Ограничения

Элемент управления Winsock действительно удобен и прост в использовании, но к сожалению, несколько ошибок делают его непригодным для кри-



тичных приложений Эти ошибки имеются в последней версии Winsock для Visual Basic 5.0, которая представляет собой новую версию элемента управления из второго пакета обновлений.

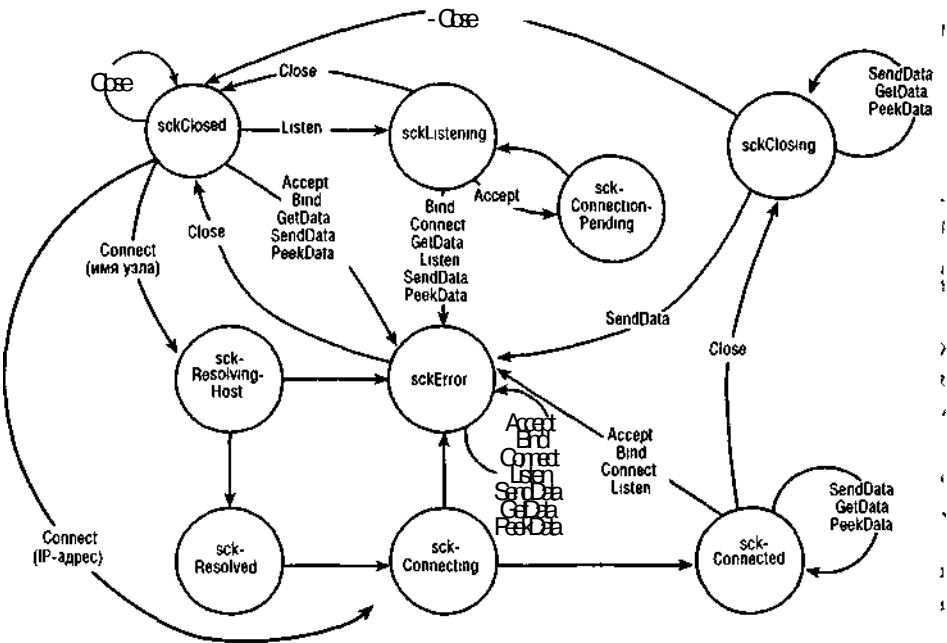


Рис. 15-4. Диаграмма состояний TCP-сокета

Первая ошибка не значительна и связана с загрузкой и выгрузкой Winsock. При выгрузке ранее загруженного элемента управления происходит утечка памяти. В связи с этим в нашем примере мы не загружаем и не выгружаем элемент управления, когда клиент устанавливает соединение и отключается. После загрузки элемента Winsock в память мы сохраняем его для использования другими клиентами.

Вторая ошибка связана с закрытием сокетного соединения, до того как в сеть будут переданы все запрошенные данные. В некоторых случаях при вызове метода *Close* после события *SendData* (когда *Close* обрабатывается раньше *SendData*) имеет место потеря данных, по крайней мере, с точки зрения получателя. Чтобы обойти эту проблему, перехватывайте событие *SendComplete* (оно наступает, после того как метод *SendData* завершит передачу данных в сеть). Кроме того, транзакции приема-передачи можно организовать так, чтобы получатель, приняв все ожидаемые данные, вызывал метод *Close* первым. В результате на принимающем компьютере наступит событие *Close*, извещающее, что все данные получены и соединение можно закрыть.

Последняя и наиболее серьезная ошибка — потеря данных, когда на передачу отправляется буфер большого размера. Если в очередь для передачи по сети поставлен достаточно большой блок данных, внутренний буфер элемента управления переполняется и некоторая информация теряется. К сожалению, полного решения этой проблемы нет. Наилучший способ — пере-

давать данные блоками, размер которых не превышает 1000 байт. Передавая буфер, подождите, пока наступит событие *SendComplete*, прежде чем передать следующий буфер с данными. Это неудобно, но значительно повышает надежность элемента управления.

В новейшей версии Winsock, поставляемой с Visual Basic 6.0, исправлены все упомянутые ошибки, за исключением второй. Если вызвать команду *Close* после вызова метода *SendData*, сокет немедленно закроется, не передав какие-либо данные. Было бы просто здорово, если бы программисты Microsoft исправили и эту ошибку, хотя она наименее серьезная и сложная.

## Типичные ошибки

Как мы уже не раз говорили, в процессе работы приложение может столкнуться с достаточно ограниченным числом ошибок Winsock. Мы не будем рассматривать их все, а обсудим лишь наиболее распространенные: *Local address in use* (Локальный адрес уже используется) и *Invalid operation at current state* (Неверное действие в текущем состоянии).

### Ошибка Local address in use

Эта ошибка наступает, если связать элемент управления с локальным портом при помощи методов *Bind* или *Connect*, при том что порт уже используется. Это наиболее распространенная ошибка TCP-сервера, который всегда связывается с конкретным портом, чтобы клиенты могли обнаружить службу.

Если перед завершением работы приложения, использующего сокет, последний не будет должным образом закрыт, он на короткий период времени перейдет в состояние *TIME\_WAIT*, чтобы гарантировать, что через этот порт отправлены (переданы) все данные. При попытке связать другой элемент Winsock с этим портом генерируется ошибка *Local address in use*.

Распространенная ошибка на стороне клиента приводит к такой же ситуации. Если свойству *LocalPort* задается значение 0 и затем устанавливается соединение, свойству *LocalPort* задается значение, соответствующее номеру локального порта, через который клиент установил соединение. Если вы собираетесь в дальнейшем установить новое соединение, используя данный элемент управления, не забудьте вернуть свойству *LocalPort* значение 0. В противном случае, если предыдущее соединение не будет корректно закрыто, возможна ошибка *Local address in use*.

### Ошибка Invalid Operation at Current State

Эта ошибка также широко распространена и возникает, если вызвать метод элемента управления Winsock, выполнение которого не допускается текущим состоянием элемента. На рис. 15-2 и 15-4 приведены диаграммы состояний UDP- и TCP-сокетов. Для создания устойчивого кода всегда проверяйте состояние сокета, прежде чем вызвать какой-либо метод.

Ошибки Winsock генерируются в результате наступления события *Error*. Это те же ошибки, что возникают при прямом программировании Winsock.

Подробное описание ошибок Winsock см. в главе 7 или в приложении С, где перечислены коды.

## Элемент управления Windows CE Winsock

В комплекте инструментов Visual Basic Toolkit for Windows CE (VBCE) имеется элемент управления Winsock, предоставляющий большинство функций «стандартного» элемента управления Winsock, поставляемого с Visual Basic. Основное его отличие — протокол UDP не поддерживается, но обеспечивается поддержка протокола IrDA. Кроме того, требуются некоторые незначительные изменения в коде приложения.

Как упоминалось в главе 7, Windows CE не поддерживает асинхронную модель Winsock, и элемент управления Windows CE Winsock не исключение из этого правила. Главное различие при программировании — метод *Connect* является блокирующим, а события *Connect* — нет. Как только вы попытаетесь установить соединение, вызвав метод *Connect*, вызов будет блокирован вплоть до установления соединения или возвращения ошибки.

Кроме того, комплект инструментов VBCE 1.0 не поддерживает массивы элементов управления, а значит, потребуется изменить код серверной части, приведенный в листинге 15-2. В результате единственный простой способ обработки нескольких соединений — разместить на форме несколько элементов управления Windows CE Winsock. На деле это ограничивает максимальное число параллельных клиентских соединений, которое приложение может обработать, поскольку такое решение вообще не масштабируется.

Кроме того, у события *ConnectionRequest* нет параметра *RequestID*, что может показаться странным. В результате придется вызвать метод *Accept* для элемента управления, которому будет передано соединение. Запрос на соединение, вызывающий событие *ConnectionRequest*, обрабатывается получающим этот запрос элементом управления.

## Пример

Рассмотрим вкратце приложение, использующее элемент управления Windows CE Winsock. Элемент Windows CE Winsock работает аналогично стандартному элементу управления Winsock, за исключением описанных различий. В листинге 15-3 приведен код, использующий элемент управления Windows CE Winsock.

### Листинг 15-3. Пример приложения, использующего элемент управления Windows CE Winsock

Option Explicit

Эта глобальная переменная позволяет сохранить текущее состояние переключателей. Значение 0 соответствует протоколу TCP, а 2 - протоколу IrDA (инфракрасный). Учтите, что на данный момент элемент управления не поддерживает протокол UDP.

Public SocketType

**Листинг 15-3.** {продолжение}

```

Private Sub cmdCloseListen_Click()
' Закройте прослушивающий сокет, и верните Я»тим кнопкам
' начальное состояние
WinSocki.Close

cmdConnect.Enabled = True
cmdListen.Enabled = True
cmdDisconnect.Enabled = False
cmdSendData.Enabled = False
cmdCloseListen.Enabled = False
End Sub

Private Sub cmdConnect_Click()
' Проверьте выбранный тип сокета и иницируйте данное соединение

If SocketType = 0 Then
' Задайте протокол, а также имя и номер порта удаленного компьютера

WinSocki.Protocol = 0
WinSocki.RemoteHost = txtServerName.Text
WinSocki.RemotePort = CInt(txtPort.Text)
WinSocki.LocalPort = 0

WinSocki.Connect
Elseif SocketType = 2 Then
' Выберите протокол IrDA и задайте имя службы

WinSocki.Protocol = 2
WinSocki.LocalPort = 0
WinSocki.ServiceName = txtServerName.Text
WinSockLRemoteHost = txtServerName.Text

WinSocki.Connect

End If
' Убедитесь, что соединение успешно установлено; вели его так,
' включите/отключите некоторые команды

MsgBox WinSocki.State
If (WinSocki.State = 7) Then
cmdConnect.Enabled = False
cmdListen.Enabled = False
cmdDisconnect.Enabled = True
cmdSendData.Enabled = True
Else
MsgBox "Connect failed"
WinSocki.Close
End If
End Sub

```

Листинг 15-3. (продолжение) ti\vt.»

```
Private Sub cmdDisconnect_Click()
```

```
    ' Закройте соединение текущего клиента
    ' и верните кнопкам начальное состояние
    WinSocki.Close
```

```
    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
```

```
End Sub
```

```
Private Sub cmdListen_Click()
```

```
    ' Переведите сокет в режим прослушивания для данного типа протокола
```

```
    If SocketType = 0 Then
        WinSocki.Protocol = 0
        WinSocki.LocalPort = CInt(txtLocalPort.Text)
        WinSocki.Listen
```

```
    ElseIf SocketType = 2 Then
        WinSocki.Protocol = 2
        WinSocki.ServiceName = txtServerName.Text
        WinSocki.Listen
```

```
    End If
```

```
    ' Если сокет находится не в режиме прослушивания,
    ' возникает ошибка
```

```
    If (WinSocki.State = 2) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdCloseListen.Enabled = True
```

```
    Else
```

```
        MsgBox "Unable to listen!"
```

```
    End If
```

```
End Sub
```

```
Private Sub cmdSendData_Click()
```

```
    ' Передайте данные из рамки по текущему соединению
```

```
    WinSocki.SendData txtSendData.Text
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Задайте начальные значения для кнопок, таймера и т.д.
```

```
    optTCP.Value = True
```

```
    SocketType = 0
```

```
    Timer1.Interval = 750
```

## Чистинг 15-3. (продолжение)

```

Timer1.Enabled = True

cmdConnect.Enabled = True
cmdListen.Enabled = True

I  cmdDisconnect.Enabled = False
II cmdSendData.Enabled = False
C  cmdCloseListen.Enabled = False

lblLocalIP.Caption = WinSocki.LocalIP
End Sub

Private Sub optIRDA_Click()
' Задайте тип сокета как IrDA
K
1  optIRDA.Value = True
   SocketType = 2
End Sub

Private Sub optTCP_Click()
1  Задайте тип сокета как TCP
   optTCP.Value = True
   SocketType = 0
   cmdConnect.Caption = "Connect"
End Sub

Private Sub Timer1_Timer()
' Это - событие, вызываемое при срабатывании таймера.
' Обновляем надпись информацией о состоянии сокета

Select Case WinSocki.State
Case 0
    lblState.Caption = "sckClosed"
Case 1
    lblState.Caption = "sckOpen"
Case 2
    lblState.Caption = "sckListening"
Case 3
    lblState.Caption = "sckConnectionPending"
Case 4
    lblState.Caption = "sckResolvingHost"
Case 5
    lblState.Caption = "sckHostResolved"
Case 6
    lblState.Caption = "sckConnecting"
Case 7
    lblState.Caption = "sckConnected"

```

## Листинг 15-3. (продолжение)

ШНУОНТ

```

Case 8
    lblState.Caption = sckClosing"
Case 9
    lblState.Caption = "sckError"
End Select
End Sub

Private Sub WinSock1_Close()
' Другая сторона инициировала закрытие соединения,
' и мы также закрываем соединение со своей стороны.
' Верните кнопкам начальное состояние

WinSock1.Close

cmdConnect.Enabled = True
cmdListen.Enabled = True

cmdDisconnect.Enabled = False
cmdSendData.Enabled = False
cmdCloseListen.Enabled = False
End Sub

Private Sub WinSock1_ConnectionRequest()
' Мы получили запрос клиента на соединение; принимаем его
' на прослушивающем сокете

WinSock1.Accept
End Sub

Private Sub WinSock1_DataArrival(ByVal bytesTotal)
' Это - событие, извещающее о поступлении данных.
' Получаем данные и добавляем их в список.

Dim rdata

WinSock1.GetData rdata
List1.AddItem rdata
End Sub

Private Sub WinSock1_Error(ByVal number, ByVal description)
    MsgBox description
    Call WinSock1_Close
End Sub

```

Мы не станем углубляться в специфику кода, поскольку он аналогичен коду приложения SockTCP из листинга 15-2. Единственное отличие — два обсуждавшихся ограничения. Как видите, элемент управления Windows CE

Winsock является достаточно «сырым», он не отработан в такой степени, как версия Winsock для настольных компьютеров. Библиотеки типов реализованы не полностью: вы должны отличать тип протокола, используя простое целочисленное значение, а не перечислимый тип. Кроме того, существует проблема с состоянием сокета (о ней — чуть позже).

Обработка инфракрасных (ИК) соединений не сильно отличается от обработки TCP-подключений. Единственное различие — когда прослушивающий сокет устанавливается на ИК-порту. ИК-сервер известен по его имени службы (подробней — в главе 6). У элемента управления Winsock имеется дополнительное свойство *ServiceName*. Ему можно задать значение из текстового поля, используя которое клиенты пытаются установить соединение. Например, следующий код переводит элемент управления Windows CE Winsock с именем *CeWinsock* в прослушивающий режим *MyServer*.

```
CeWinsock.Protocol = 2          ' протокол 2 - IrSock
CeWinsock.ServiceName = "MyServer"
CeWinsock.Listen
```

Других требований к публикации службы с использованием ИК-сокетов нет. Вам необходимо указать лишь имя службы.

## Проблема с элементом управления VBCE Winsock

Не совсем понятно использование перечислимых значений для состояний Winsock. По какой-то странной причине эти значения определяются в среде разработки, однако при использовании перечислимого типа *sock* в коде на удаленных устройствах постоянно выщется сообщение *An error was encountered while running this program*. Если заменить перечислимые типы соответствующими числовыми эквивалентами, ошибка исчезает. Это считается неполадкой и будет исправлено в последующих версиях комплекта разработчика.

## Резюме

Элемент управления Winsock, поставляемый с Visual Basic, полезен в простых, некритичных приложениях, которым необходимы функции работы с сетью. Некоторые проблемы в версии Winsock, поставлявшейся с Visual Basic 5.0, затрудняют использование элемента управления в приложениях. В последней версии Visual Basic все они устранены. Winsock позволяет добавить в приложение Visual Basic простые функции работы с сетью. Конечно, возможности элемента управления ограничены, и для приложений, которым требуется интенсивно взаимодействовать с Winsock, лучше вручную импортировать все функции и константы из DLL-библиотеки Winsock. Как уже упоминалось, во второй части книги приведены примеры на Visual Basic, импортирующие функции Winsock из библиотеки *Ws2\_32.dll*. Эти приложения — *SimpleTCP* и *SimpleUDP* расположены на прилагаемом компакт-диске, как и соответствующие файлы *Winsock.bas*.



# СЛУЖБА УДАЛЕННОГО ДОСТУПА (RAS)

UK

I.O.

Мы уже описали все доступные в Microsoft Windows сетевые API-функции, которые позволяют разрабатывать приложения, способные взаимодействовать с другими приложениями по сети. В большинстве случаев наши исследования фокусировались на приложениях и уровнях представления данных семиуровневой модели OSI. Мы не обсуждали детали отдельных сетевых протоколов и в основном рассматривали использование функций, независимо от протокола.

Последняя часть книги посвящена *службе удаленного доступа* (Remote Access Service, RAS), позволяющей пользователям подключаться к локальной сети с удаленного компьютера и использовать сетевые функции, как если бы они были подключены к удаленной сети напрямую.

## Клиент службы RAS

Все платформы Microsoft Windows используют клиент RAS, позволяющий подключаться к удаленному компьютеру, если тот является сервером удаленного доступа. Обычно клиент RAS задействует для этого модем, который подсоединяется к телефонной линии и вызывает удаленный компьютер путем набора номера. Из-за этого клиент RAS иногда называют *клиентом удаленного доступа* (dial-up networking, DUN).

Сервер должен находиться в режиме ожидания вашего DUN-подключения. Клиент RAS может устанавливать соединение с несколькими типами серверов удаленного доступа. Для этого он пользуется стандартными промышленными *кадрирующими* (framing) протоколами, которые могут передавать следующие протоколы связи.

**III Point-to-Point Protocol (PPP) - IP, IPX и NetBEUI;**

- **Serial Line Internet Protocol (SLIP)** — только IP;
- **Asynchronous NetBEUI (Microsoft Windows NT 3.1, Microsoft Windows for Workgroups 3-11)** — только NetBEUI.

Кадрирующие протоколы описывают процесс передачи данных через RAS-соединение и указывают, какой протокол сетевого соединения (типа TCP/IP или IPX) может устанавливать связь через подключение. Если сервер поддерживает один из перечисленных кадрирующих протоколов, то клиент может установить соединение. В Windows 95, 98, 2000 и NT предусмотрен компонент сервера RAS, поддерживающий все указанные кадрирующие протоколы.

Если было установлено соединение между RAS-клиентом и сервером, то стеки сетевого протокола (в зависимости от используемого кадрирующего протокола) могут взаимодействовать через подключение RAS с удаленным компьютером, как если бы компьютеры были соединены в рамках ЛВС. Естественно, пропускная способность многих модемов сегодня значительно ниже прямого соединения ЛВС.

Когда сервер RAS принимает соединение по телефонной линии, то сначала он устанавливает соединение с клиентом с помощью одного из кадрирующих протоколов. Если кадрирующий протокол определен, то сервер делает попытку аутентифицировать пользователя, устанавливающего соединение. API-функции RAS позволяют клиенту задать имя пользователя, пароль и домен, проверяющий учетные реквизиты на сервере. Когда RAS-сервер Windows 2000 или Windows NT получает эту информацию, он подтвержда-

\* ет учетные реквизиты, применяя доменные механизмы безопасности доступа. При этом сервер RAS не подключает клиента к домену Windows NT, а проверяет с помощью реквизитов права пользователя на подключение.

Процесс подключения RAS не то же самое, что процесс входа в домен Windows NT, но после успешного соединения удаленный компьютер может войти в домен Windows NT. В этой главе вы не найдете описания процесса подключения к домену. В Windows 95 и Windows 98 служба RAS способна автоматически произвести вход в домен от имени компьютера после аутентификации удаленного соединения через телефонный справочник.

Для установки и управления устройствами связи, такими как модемы, используется *программный интерфейс компьютерной телефонии* (Telephony Application Programming Interface, TAPI). Он управляет аппаратными параметрами этих устройств. Когда вы устанавливаете RAS-соединение с помощью модема, TAPI включает модем и посылает ему информацию о набираемом номере. В результате RAS рассматривает модемы просто как модемные порты TAPI-интерфейса, которые могут набирать номер и устанавливать телефонное соединение с удаленным сервером. Некоторые из API-функций RAS ссылаются на модемные порты TAPI-интерфейса, когда вы запрашиваете информацию о RAS-соединении.

В этой главе мы расскажем, как программно использовать RAS для установления соединений с удаленной сетью. Начнем с описания файлов заголовков и библиотек, необходимых для создания приложения. Затем будут описаны основы соединений через телефонную линию — как фактически устанавливается удаленное соединение. Далее мы рассмотрим настройку записей телефонного справочника RAS для определения детальных свойств RAS-соединения. И наконец, поговорим о том, как управлять уже установленным соединением.

## Компиляция и компоновка

При разработке приложения RAS нужно подключить файлы заголовков и библиотек:

- **Ras.h** — содержит прототипы функций и структуры данных, используемые API-функциями RAS;
- **Raserror.h** — содержит заданные коды ошибок, возвращаемые API-функциями RAS при сбоях;

И **Rasapi32.lib** — библиотека всех API-функций RAS.

В файле Raserror.h вы найдете строку описания ошибки для каждого кода, используемого в RAS. Функция *RasGetErrorString* позволяет программно отыскивать строки с описанием ошибки, ассоциируемые с конкретным кодом:

```
DWORD RasGetErrorString(  
    UINT uErrorValue,  
    LPTSTR lpszErrorString,  
    DWORD cBufSize
```

Параметр *uErrorValue* получает конкретное значение кода ошибки RAS возвращаемое RAS-функцией. Параметр *ipzErrorString* — это предоставляемый приложением буфер, куда будет занесено описание ошибки с кодом из *uErrorValue*. Буфер для хранения этой строки нужно сделать достаточно большим — не менее 256 символов. Иначе процедура вернет ошибку *ERROR\_INSUFFICIENT\_BUFFER*. Последний параметр — *cBufSize*, содержит размер буфера, который вы задали в *IpzErrorString*.

## Структуры данных и вопросы совместимости платформ

Используемые RAS-функциями структуры данных имеют дополнительные поля данных: включенные или нет — зависит от значения *WINVER*. В структуре данных RAS также есть поле *dwSize*, в котором необходимо задать размер используемой вами RAS-структуры в байтах. Это влияет на работу RAS-функций, использующих эти структуры, так как они настроены на конкретную платформу. Правила *WINVER* применимы для платформ Windows 95, 98, 2000 и NT, но не для Windows CE (Windows CE SDK не определяет *WINVER*). Следующие параметры указывают, что приложение RAS настроено под:

- **WINVER = 0x400** - Windows 95, 98 или NT 4; и, 1.
- III **WINVER = 0x401** — Windows NT 4; " "
- **WINVER = 0x500** - Windows 2000. • 3йОШГх3.Ша

Служба RAS не рассчитана на использование единственного исполняемого файла, запускающегося на всех платформах. Теоретически, при аккуратном программировании можно обеспечить работу приложения на всех платформах (конечно, исключая Windows CE). Но мы все же рекомендуем конструировать RAS-приложение для конкретной платформы.

## Обновление DUN 1.3 и Windows 95

Между исходной версией и OSR 2 было выпущено еще несколько версий Windows 95. OSR 2 не была конечным продуктом, напротив, она устанавливалась только на новые OEM-компьютеры. У каждой версии свои особенности работы API-функций RAS, поэтому рекомендуется установить последний пакет обновления RAS — DUN 1.3, загрузив его со страницы <http://www.microsoft.com/support>. Но мы предполагаем, что вы это уже сделали, и не будем обсуждать RAS из предыдущих версий DUN.

## Функция *RasDial*

Для подключения к удаленному компьютеру RAS-приложение клиента вызывает функцию *RasDial*. Это довольно сложная функция с множеством параметров вызова, используемых для набора номера, аутентификации и обновления соединения с удаленным сервером. *RasDial* определяется следующим образом:

```

DWORD RasDialK
    LPRASDIALEXTENSIONS lpRasDialExtensions,
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lpRasDialParams,
    DWORD dwNotifierType,
    LPVOID lpvNotifier,
    LPHRASCONN lphRasConn
);

```

Параметр *lpRasDialExtensions* — необязательный указатель на структуру *RASDIALEXTENSIONS*, которая позволяет приложению активизировать расширенные возможности для *RasDial*. В Windows 95, 98 и CE этот параметр игнорируется и должен иметь значение *NULL*. Структура *RASDIALEXTENSIONS* определена так

```

typedef struct tagRASDIALEXTENSIONS {
    DWORD      dwSize,                                     I
    DWORD      dwfOptions,
    HWND       hwndParent;                                i
    ULONG_PTR  reserved;
#ifdef WINVER >= 0x500
    ULONG_PTR  reserved2,
    RASEAPIINFO RasEapInfo,
#endif
} RASDIALEXTENSIONS,

```

Как мы и говорили, размер этой структуры на этапе компиляции зависит от значения *WINVER*. Ее поля описываются следующим образом

- ***dwSize*** — должен содержать размер (в байтах) структуры *RASDIALEXTENSIONS*

**III *dwfOptions*** — позволяет задавать для использования расширений *RasDial* следующие битовые флаги

8 *RDEOPTJJsePrefixSuffix* — задает для *RasDial* использование префикса и суффикса, ассоциированных с конкретным устройством набора номера,

И *RDEOPT\_PausedStates* — дает возможность *RasDial* использовать режим паузы, позволяя пользователям производить повторный вход в систему, изменять пароли и задавать номер обратного вызова,

III *RDEOPT\_IgnoreModemSpeaker* — *RasDial* игнорирует параметры динамика модема в телефонном справочнике RAS,

И *RDEOPT\_SetModemSpeaker* — включает динамик модема, если задан *RDEOPT\_IgnoreModemSpeaker*,

III *RDEOPT\_IgnoreSoftwareCompression* — *RasDial* игнорирует параметры программного сжатия,

И *RDEOPT\_SetSoftwareCompression* — включает программное сжатие, если задан *RDEOPT\_IgnoreSoftwareCompression*,

III *RDEOPT\_PauseOnScript* — предназначен для внутреннего использования *RasDialDlg*, вам не нужно задавать этот флаг,

III *bwndParent* — не используется и должен быть равен *NULL*

III *reserved* — не используется и должен иметь значение 0

III *reserved1* — зарезервирован для будущего использования в Windows 2000 и должен иметь значение 0

- *RasEapInfo* — в Windows 2000 позволяет указать информацию протокола Extensible Authentication Protocol (EAP)

В *RasDial* параметр *ipaszPhonebook* идентифицирует путь к файлу телефонного справочника в Windows 2000 и NT В Windows 95, 98 и SE этот параметр должен иметь значение *NULL*, так как телефонный справочник хранится в системном реестре *Телефонный справочник* (phonebook) — это совокупность свойств набора номера RAS, определяющих способ установки RAS-соединения *RasDial* использует достаточно много свойств набора номера. Подробнее мы поговорим об этом позже.

Указатель *ipRasDialParams* структуры *RASDIALPARAMS* определяет параметры набора номера и аутентификации пользователя, которые использует функция *RasDial* для установления удаленного соединения.

```
typedef struct .RASDIALPARAMS {
    DWORD dwSize,
    TCHAR szEntryName[RAS_MaxEntryName + 1],
    TCHAR szPhoneNumber[RAS_MaxPhoneNumber + 1],
    TCHAR szCallbackNumber[RAS_MaxCallbackNumber + 1];
    TCHAR szUserName[UNLEN + 1],
    TCHAR szPassword[PWLEN + 1],
    TCHAR szDomain[DNLEN + 1] ;
#ifdef WINVER >= 0x401
    DWORD dwSubEntry,
    DWORD dwCallbackId;
#endif
} RASDIALPARAMS,
```

*RASDIALPARAMS* имеет следующие поля

- *dwSize* — размер структуры *RASDIALPARAMS* в байтах. Это позволяет RAS определять версию *WINVER*, с которой вы работаете.

III *szEntryName* — строка, позволяющая идентифицировать запись в телефонном справочнике (содержится в файле телефонного справочника, указанном в параметре *IpszPhonebook* функции *RasDial*). Как мы уже говорили, записи в телефонном справочнике дают возможность точно настраивать такие свойства RAS-соединения, как выбор модема или кадрирующего протокола. Однако указывать запись телефонного справочника для использования *RasDial* не обязательно. Если это поле — пустая строка («»), *RasDial* выберет первый доступный модем, установленный в системе, и с помощью следующего параметра — *szPhoneNumber*, установит соединение.

- *szPbnumber* — телефонный номер, который переопределяет номер из записи телефонного справочника, указанной в поле *szEntryName*.

**Я** *szCallbackNumbei* — телефонный номер, по которому RAS-сервер может установить с клиентом обратную связь. В этом случае сервер разорвет первоначальное соединение и сам установит связь с клиентом по указанному номеру. Это удобное свойство позволяет серверу узнать, что за пользователь к нему подсоединяется.

**III** *szUserName* — идентифицирует имя пользователя, используемое для его аутентификации на сервере.

**И** *szPassword* — идентифицирует пароль, используемый для аутентификации пользователя на сервере.

**II** *szDomain* — идентифицирует домен Windows 2000 или Windows NT, на котором размещена учетная запись пользователя.

- *dwSubEntry* — дополнительно позволяет указывать первоначальную подзапись телефонного справочника для подключения к многоканальному соединению RAS (будут рассмотрены далее).

**III** *dwCallbackId* — позволяет передавать определенное приложением значение в функцию обратного вызова *RasDialFunc2* (которая тоже будет описана позже). Если вы не используете эту функцию обратного вызова, то данное поле не применяется.

Следующие параметры — *dwNotifierType* и *IpnNotifier*, определяют режим работы *RasDial* (синхронно или асинхронно она вызывается). Последний параметр — *iphRasConn*, указывает на описатель RAS-соединения типа *HRASCONN*. Перед вызовом *RasDial* нужно присвоить этому параметру значение *NULL*. Если *RasDial* выполняется успешно, то возвращается ссылка на описатель RAS-соединения.

Теперь рассмотрим, как вызывается *RasDial*. Как уже упоминалось, у *RasDial* два режима работы: синхронный и асинхронный. В синхронном режиме *RasDial* блокируется, пока не будет завершена или отклонена установка соединения. В асинхронном режиме *RasDial* завершает установку соединения сразу, позволяя приложению выполнять в это время другие действия.

**М**

## Синхронный режим

Если параметр *IpnNotifier* функции *RasDial* равен *NULL*, *RasDial* будет работать синхронно, а параметр *dwNotifierType* игнорируется. Это наиболее простой способ работы с *RasDial*, однако в этом случае вы не сможете отслеживать соединение, как в асинхронном режиме. Листинг 16-1 показывает, как вызвать *RasDial* в синхронном режиме. Заметьте, что этот код не задает телефонный справочник или запись справочника — он лишь демонстрирует, как сформировать RAS-соединение.

### Листинг 16-1. Синхронный вызов *RasDial*

```
RASDIALPARAMS RasDialParams;
HRASCONN hRasConn;
DWORD Ret;
```

IV

ff

II Всегда задавайте размер структуры *RASDIALPARAMS*

**Листинг 16-1.** (продолжение)

```

RasDialParams.dwSize = sizeof(RASDIALPARAMS);
hRasConn = NULL;

// Если записать в этом поле пустую строку, RasDial будет
// использовать стандартные свойства установки связи.

lstrcpy(RasDialParams.szEntryName,          " ");

lstrcpy(RasDialParams.szPhoneNumber, "867-5309");
lstrcpy(RasDialParams.szUserName,  "jenny");
lstrcpy(RasDialParams.szPassword, "mypassword");
lstrcpy(RasDialParams.szDomain, "mydomain");

// Вызов RasDial в синхронном режиме
// (пятый параметр равен NULL)
Ret = RasDial(NULL, NULL, &RasDialParams, 0, NULL, ihRasConn);

if (Ret != 0)

    printf("RasDial failed: Error = %d\n", Ret);

```

**Асинхронный режим**

Асинхронный вызов *RasDial* гораздо сложнее. Если параметр *IpvNotifier* функции *RasDial* не равен *NULL*, то *RasDial* будет работать в асинхронном режиме, то есть запросы выполняются сразу, и соединения продолжают устанавливаться. Вызов *RasDial* в асинхронном режиме предпочтительнее для установления RAS-соединения, так как позволяет отслеживать ход подключения. Параметр *IpvNotifier* может быть как указателем на функцию, вызываемую, когда *RasDial* выполняет подключение, так и описателем окна, получающего уведомления о ходе выполнения через сообщения Windows. Параметр *dwNotifierType* функции *RasDial* определяет тип функции или описателя окна, переданного в *IpvNotifier*. Вы можете указывать в *dwNotifierType* для параметра *IpvNotifier* следующие значения:

- III 0 — для управления событиями подключения *RasDial* будет использовать указатель на функцию *RasDialFunc*;
- III 1 — для управления событиями подключения *RasDial* будет использовать указатель на функцию *RasDialFunc1*;
- III 2 — для управления событиями подключения *RasDial* будет использовать указатель на функцию *RasDialFunc2*;
- III 0xFFFFFFFF — *RasDial* должна отправить оконное сообщение в ходе подключения.

Иными словами, есть три прототипа функций, которые вы можете передавать в параметр *IpvNotifier* функции *RasDial* для получения обратных уве-



домлений о событиях подключения *RasDialFunc*, *RasDialFund* и *RasDialFunc2* Первый — *RasDialFunc*, описан так

```
VOID WINAPI RasDialFunc(
    UINT unMsg,
    RASCONNSTATE rasconnstate,
    DWORD dwError
```

В параметр *unMsg* передается тип произошедшего события. На данный момент таким событием может быть только *WM\_RASDIALEVENT*, а значит, этот параметр бесполезен. В параметр *rasconnstate* передается активность подключения, которая будет начата функцией *RasDial*. Табл. 16-1 описывает возможные действия подключения. Если одно из них не выполняется, в параметр *dwError* передается код ошибки RAS.

**Табл. 16-1. Действия подключения RAS**

Состояние	Действие	Описание
Выполнение	<i>RASCSOpenPort</i>	Коммуникационный порт готовится к открытию
	<i>RASCS_PortOpened</i>	Коммуникационный порт открыт
	<i>RASCS_ConnectDevice</i>	Устройство готово к подключению
	<i>RASCSJDevweConnected</i>	Устройство успешно подключено
	<i>RASCS_AUDevices-Connected</i>	Установлен физический канал передачи данных
	<i>RASCS_Authenticate</i>	Процесс аутентификации RAS начат
	<i>RASCS_AuthNotiJy</i>	Произошло событие аутентификации
	<i>RASCS_AuthRetry</i>	Клиент сделал запрос на новую попытку аутентификации
	<i>RASCS_AuthCallback</i>	Сервер запросил номер обратного вызова
	<i>RASCS_AuthChange-Password</i>	Клиент сделал запрос на изменение пароля в учетной записи
	<i>RASCS_AuthProject</i>	Проекция протокола начата
	<i>RASCS_AuthLmkSpeed</i>	Вычисляется скорость соединения
	<i>RASCS_AuthAck</i>	Запрос на аутентификацию подтверждается
	<i>RASCS_ReAuthenticate</i>	Процесс аутентификации будет начат после обратного вызова
	<i>RASCS_Authenticated</i>	Клиент успешно завершил аутентификацию
	<i>RASCS_PrepateFor-Callback</i>	Линия готова к отключению для подготовки к обратному вызову
	<i>RASCS_WmtFor-ModemReset</i>	Клиент ожидает перезагрузки модема для подготовки к обратному вызову
	<i>RASCS_WaitForCallback</i>	Клиент ожидает входящего звонка от сервера
	<i>RASCS_Projected</i>	Проекция протокола завершена
	<i>RASCS_StartAuthen-tication</i>	Аутентификация пользователя начинается или повторяется (только для Windows 9x)

Табл. 16-1. (продолжение)

Состояние	Действие	Описание
	<i>RASCS_CallbackComplete</i>	До клиента был произведен обратный вызов (только для Windows 9x)
	<i>RASCS_LogonNetwork</i>	Клиент подключается к удаленной сети (только для Windows 9x)
	<i>RASCSJubEntry-Connected</i>	Подключена подзапись многоканальной записи телефонного справочника. Параметр <i>dwSubEntry</i> функции <i>RasDiaWunc2</i> получает значение индекса подсоединенной подзаписи
	<i>RASCSJubEntryDts-connected</i>	Подзапись записи многоканального телефонного справочника отключилась. Параметр <i>dwSubEntry</i> функции <i>RasDialFunc2</i> получает значение индекса отсоединенной подзаписи
Пауза	<i>RASCS_RetryAuthentication</i>	<i>RasDial</i> в режиме ожидания учетных реквизитов нового пользователя
	<i>RASCS_CallbackSetBy-Caller</i>	<i>RasDial</i> ожидает от клиента номера обратного вызова
	<i>RASCSJPasswordExpired</i>	<i>RasDial</i> ожидает от пользователя задания нового пароля
Завершение	<i>RASCSJInvokeEapUI</i>	В Windows 2000 <i>RasDial</i> ожидает, пока специальный пользовательский интерфейс получит EAP-информацию
	<i>RASCSJConnected</i>	RAS-соединение успешно установлено и активно
	<i>RASCS_Disconnected</i>	RAS-соединение установить не удалось или оно неактивно

Как видно из табл. 16-1, существуют три режима работы, ассоциируемые с действиями подключения при асинхронном вызове *RasDial*: выполнение, пауза и завершение. Выполнение означает, что *RasDial* в процессе работы, и каждый процесс предлагает информацию о ходе работы.

Пауза означает, что *RasDial* требуется дополнительная информация для установления соединения. По умолчанию режим паузы отключен. Вы можете включить его, задав флаг *RDEOPT\_PausedStates* в структуре *RASDIALEXTENSIONS*. Когда система в режиме паузы, это может означать, что пользователь должен:

- III ввести новые учетные реквизиты, так как аутентификация не удалась,
- III ввести новый пароль, так как текущий устарел,
- III ввести номер обратного вызова

Эти параметры активности относятся к информации в структуре *RASDIAL\_PARAMS*, описанной ранее. Когда возникнет пауза, *RasDial* уведомит об этом клиентскую функцию обратного вызова (или оконную процедуру). Если режим паузы отключен, то RAS отправит ошибку в клиентскую функцию уведомления, и *RasDial* даст сбой, если включен — *RasDial* останется в режиме,

позволяющем приложению вводить новую информацию через структуру *RASDIALPARAMS*. Когда *RasDial* в режиме паузы, вы можете возобновить ее работу, вызвав ее заново с описателем соединения исходного вызова *QphRasConri* и функцией уведомления (*ipvNotifier*). Или просто завершите работу в режиме паузы, вызвав *RasHangUp* (описана далее в этой главе). Если вы возобновляете приостановленное соединение, то задайте входные данные пользователя через структуру *RASDIALPARAMS*, переданную в функцию *RasDial* при возобновляемом вызове.

**ПРИМЕЧАНИЕ** Не возобновляйте работу *RasDial* путем вызова ее прямо из функции обработчика уведомления (например, *RasDialFunc*). *RasDial* разработана так, что не поддерживает данную ситуацию, так что лучше возобновить работу *RasDial* прямо из потока приложения.

Последний режим — завершение, свидетельствует, либо что соединение успешно установлено, либо что его установить не удалось, либо что функция *RasHangUp* закрыла соединение.

Теперь продемонстрируем простую программу, асинхронно вызывающую *RasDial* (листинг 16-2). Пример асинхронного вызова *RasDial* вы также найдете на прилагаемом компакт-диске.

#### Листинг 16-2. Асинхронный вызов *RasDial*

```
void main(void)
{
    DWORD Ret;
    RASDIALPARAMS RasDialParams;
    HRASCONN hRasConn;

    // Задайте в структуре RASDIALPARAMS параметры вызова,
    // как это было сделано в примере синхронного вызова

    if ((Ret = RasDial(NULL, NULL, &RasDialParams, 0,
        &RasDialFunc, &hRasConn)) != 0)
    {
        printf("RasDial failed with error Xd\n", Ret);
        return;

        // Выполните другие задачи, пока работает RasDial

    // Функция обратного вызова RasDialFunc()
    void WINAPI RasDialFunc(UINT unMsg, RASCONNSTATE rasconnstate,
        DWORD dwError)
    {
        char szRasString[256]; // Буфер для строки с ошибкой

        if (dwError)
```

**Листинг 16-2.** {продолжение}

```

{
    RasGetErrorString((UINT)dwError, szRasString, 256);
    printf("Error: Xd - Xs\n",dwError, szRasString);
    return;

    // Привязка каждого из режимов RasDial и вывод на экран
    // сведений о состоянии, в которое переходит RasDial

Л
    switch (rasconnstate)

*к      case RASCS_ConnectDevice:
-д^i      printf ("Connecting device.. ,\n");
.<\\<      break;
ф      case RASCS_DeviceConnected:
.^      printf ("Device connected.\n");
j,      break;

*      // Здесь можно добавить другие действия подключения

-Он
-1^,      default:
          printf ("Unmonitored RAS activity.\n");
          break;

```

Ранее мы также упоминали о других функциях обратного вызова для уведомления — *RasDialFunc1* и *RasDialFunc2*:

```

VOID WINAPI RasDialFuncK
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rases,
    DWORD dwError,
    DWORD dwExtendedError

DWORD WINAPI RasDialFunc2(
    DWORD dwCallbackId,
    DWORD dwSubEntry,
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rases,
    DWORD dwError,
    DWORD dwExtendedError
);

```

Функция *RasDialFunc1* похожа на *RasDialFunc*, рассмотренную ранее. Отличие в том, что у *RasDialFunc1* два дополнительных параметра: *hrasconn*

и *dwExtendedError*. Параметр *hrasconn* — это описатель возвращаемого *RasDial* соединения. Параметр *dwExtendedError* позволяет получить расширенную информацию об ошибке. Иногда во время подключения происходят следующие типы ошибок:

- **ERROR\_SERVERNOTRESPONDING** или **ERROR\_NETBIOSERROR** — *dwExtendedError* получает код ошибки, определенной NetBIOS;
- III **ERRORAUTHINTERNAL** — *dwExtendedError* получает код ошибки внутренней диагностики (эти коды не документированы);
- В **ERRORCANNOTGETIANA** — *dwExtendedError* получает код ошибки, определяемой маршрутизируемой RAS.

Функция *RasDialFunc2* аналогична *RasDialFunc1*, но у нее еще два параметра: *dwCallbackId* и *dwSubEntry*. Параметр *dwCallbackId* содержит определяемое приложением значение, которое первоначально было в поле *dwCallbackId* структуры *RASDIALPARAMS*, переданной в вызов *RasDial*. В параметр *dwSubEntry* передается индекс подзаписи телефонного справочника, по которому *RasDialFunc2* осуществила обратный вызов.

## Уведомление о состоянии

RAS использует автономную функцию *RasConnectionNotification*, позволяющую приложению прекращать работу во время создания или закрытия асинхронного RAS-соединения:

```
DWORD RasConnectionNotification(
    HRASCONN hrasconn,
    HANDLE hEvent,
    DWORD dwFlags
);
```

Параметр *hrasconn* — это описатель соединения, возвращенный *RasDial*. Параметр *hEvent* — описатель события, который создается приложением с использованием функции *CreateEvent*. Значением параметра *dwFlags* может быть комбинация следующих флагов действий подключения:

- III **RASCN\_Connection** — уведомляет о создании RAS-соединения; если параметр *hrasconn* равен *INVALID\_HANDLE\_VALUE*, событие освобождается при каждом RAS-соединении;
- III **RASCN\_Disconnection** — уведомляет о завершении RAS-соединения; если параметр *hrasconn* равен *INVALID\_HANDLE\_VALUE*, событие освобождается при каждом завершении RAS-соединения;
- В **RASCN\_BandwidthAdded** — при многоканальном соединении событие освобождается, когда подсоединяется подзапись;
- III **RASCN\_BandwidthRemoved** — при многоканальном соединении событие освобождается, когда отсоединяется подзапись.

Заметьте, что эти флаги функционируют аналогично флагам действий подключения (табл. 16-1). Если какие-то из событий происходят во время подключения, событие будет помечено как *свободное* (signaled). Затем, что-

бы узнать об освобождении объекта, приложение должно использовать Win32-4)уНКУНН ожидания, типа *WaitForSingleObject*

## Завершение соединения

Завершить установленное соединение с помощью *RasDial* просто. Нужно лишь вызвать функцию *RasHangUp*:

```
DWORD RasHangUp(
    HRASCONN hrasconn
```

Параметр *hrasconn* — это описатель, возвращаемый *RasDial*. Функция проста, но при ее использовании все же нужно учитывать внутреннюю организацию соединения в RAS. Соединение использует модемный порт, и порту требуется время для внутреннего возврата в исходное состояние, когда соединение закрывается. Так что нужно подождать окончательного закрытия соединения через порт. Чтобы узнать состояние сброшенного соединения, вызовите *RasGetConnectionStatus*:

```
DWORD RasGetConnectStatus(
    HRASCONN hrasconn,
    LPRASCONNSTATUS lprasconnstatus
);
```

Параметр *hrasconn* — это описатель, возвращаемый *RasDial*. Параметр *lprasconnstatus* — структура *RASCONNSTATUS*, получающая состояние текущего соединения:

```
typedef struct .RASCONNSTATUS
{
    DWORD dwSize;
    RASCONNSTATE rasconnstate;
    DWORD dwError;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASCONNSTATUS;
```

Перечисленные в отрывке кода поля определены следующим образом:

- III *dwSize* — размер (в байтах) структуры *RASCONNSTATUS*;
- III *rasconnstate* — один из параметров активности подключения (табл. 16-1);
- III *dwError* — конкретный код ошибки RAS, если *RasGetConnectStatus* не возвращает 0;
- III *szDeviceType* — тип устройства, используемого при соединении.
- III *szDeviceName* — имя текущего устройства.

Рекомендуется проверять состояние соединения, пока не будет получено значение *RASCS\_Disconnected*. Очевидно, придется несколько раз вызывать *RasGetConnectionStatus*, прежде чем соединение закроется. Затем можете выйти из приложения или установить новое соединение.

## Телефонный справочник

RAS может устанавливать соединение с удаленным сервером, используя записи телефонного справочника для хранения и организации свойств Телефонный справочник — это совокупность структур *RASENTRY*, в которых содержатся телефонные номера, скорость передачи данных, информация для аутентификации пользователя и др. В Windows 95, 98 и CE телефонный справочник хранится в системном реестре, в Windows 2000 и NT — в файлах, обычно имеющих расширение *pbk*. Структура *RASENTRY* определена так

```
typedef struct tagRASENTRY
{
    DWORD        dwSize,                                f-»
    DWORD        dwfOptions,                             Ikt.
    DWORD        dwCountryId,                            iqi
    DWORD        dwCountryCode,                          ЦД >
    TCHAR        szAreaCode[RAS_MaxAreaCode + 1],        ' 0)
    TCHAR        szLocalPhoneNumber[RAS_MaxPhoneNumber + 1],
    DWORD        dwAlternateOffset,                       #
    RASIPADDR    lpaddr,                                  'III
    RASIPADDR    lpaddrDns,
    RASIPADDR    lpaddrDnsAlt,
    RASIPADDR    lpaddrWins,                              . (
    RASIPADDR    lpaddrWinsAlt,
    DWORD        dwFrameSize,
    DWORD        dwfNetProtocols,
    DWORD        dwFramingProtocol,                       h
    TCHAR        szScript[MAX_PATH],                      Г О Т Ш Ю Ш М Я .
    TCHAR        szAutodialDll[MAX_PATH],
    TCHAR        szAutodialFunc[MAX_PATH],
    TCHAR        szDeviceType[RAS_MaxDeviceType + 1],
    TCHAR        szDeviceName[RAS_MaxDeviceName + 1],
    TCHAR        szX25PadType[RAS_MaxPadType + 1],
    TCHAR        szX25Address[RAS_MaxX25Address + 1],
    TCHAR        szX25Facilities[RAS_MaxFacilities + 1],
    TCHAR        szX25UserData[RAS_MaxUserData + 1].
    DWORD        dwChannels,
    DWORD        dwReserved1,
    DWORD        dwReserved2,
#ifdef (WINVER >= 0x401)
    DWORD        dwSubEntries,
    DWORD        dwDialMode,
    DWORD        dwDialExtraPercent,
    DWORD        dwDialExtraSampleSeconds,
    DWORD        dwHangllpExtraPercent,
    DWORD        dwHangUpExtraSampleSeconds,
    DWORD        dwIdleDisconnectSeconds,
#endif
}
#endif
#ifdef (WINVER >= 0x500)
```

```

DWORD    dwType,
DWORD    dwEncryptionType,
DWORD    dwCustomAuthKey,
GUID      guidId
TCHAR    szCustomDialDll[MAX_PATH],
DWORD    dwVpnStrategy,
#endif
> RASENTRY,

```

Ниже перечислены поля этой структуры

**III dwSize** — определяет размер (в байтах) структуры *RASENTRY*

**III dwfOptions** — может иметь значение одного или нескольких необязательных флагов

- *RASEO\_Custom* — применяется пользовательское шифрование (в Windows 2000),

**III RASEO\_DisableLcpExtensions** — RAS отключает определенные в RFC 1570 расширения PPP LCP,

**III RASEOJpHeaderCompression** — RAS согласует сжатие заголовка IP через PPP-соединение,

- *RASEOJAodemLights* — в Windows 2000 панель задач отобразит монитор состояния,

**III RASEOJNetworkLogon** — в Windows 9x RAS попытается подключить пользователя к домену только после аутентификации RAS-соединения,

**III RASEOJPreviewDomam** — в Windows 2000 RAS отобразит домен пользователя до вызова,

**В RASEO\_PrevwwPhoneNumber** — в Windows 2000 RAS отобразит телефонный номер для вызова,

**III RASEO\_PromoteAlternates** — RAS может сделать главным дополнительный номер, если по нему успешно устанавливается соединение,

**Я RASEO\_RemoteDefaultGateway** — когда активно RAS-соединение, маршрут для IP-пакетов по умолчанию предпочтительнее устанавливается через адаптер удаленной связи, а не через другой сетевой адаптер,

- *RASEO\_RequireCHAP* — в Windows 2000 для аутентификации будет использован протокол Challenge Handshake Authentication Protocol (CHAP),

**III RASEO\_RequireDataEncryphon** — шифрование данных должно быть успешно согласовано, иначе соединение следует разорвать, при этом также следует включить флаг *RASEO\_RequireEncryptedPw*,

**III RASEO\_JZequireEAP** — в Windows 2000 для аутентификации будет использован EAP,

**III RASEO\_ReqmreEncryptedPw** — не позволяет PPP использовать протокол Password Authentication Protocol (PAP) для аутентификации клиента, вместо этого применяются протоколы CHAP и Shiva's Password Authentication Protocol (SPAP),



- Я *RASEO\_RequireMsCHAP* — в Windows 2000 для аутентификации будет использован протокол Microsoft CHAP;
- *RASEO\_RequireMsCHAP2* — в Windows 2000 для аутентификации будет использован протокол Microsoft CHAP версии 2;
- *RASEO\_RequireMsEncryptedPw* — перекрывает *RASEO\_RequireEncryptedPw* и позволяет RAS использовать схемы защиты паролем, такие как Microsoft CHAP;
- II *RASEOJRequirePAP* — в Windows 2000 для аутентификации будет использован протокол PAP;
- Я *RASEO\_RequireSPAP* — в Windows 2000 для аутентификации будет использован протокол SPAP;
- III *RASEO\_RequireW95MSCHAP* — в Windows 2000 для аутентификации будет использована старая версия протокола Microsoft SPAP (разработанная для RAS-сервера Windows 95);
- III *RASEO\_ReviewUserPW* — в Windows 2000 RAS отобразит имя пользователя и его пароль до вызова;
- *RASEO\_SecureLocalFiles* — в Windows 2000 и Windows NT RAS проверит существование удаленной файловой системы и привязки удаленных принтеров до установления соединения;
- *RASEO\_SharedPhoneNumbers* — в Windows 2000 телефонные номера будут использоваться совместно;
- *RASEOjShowDialingProgress* — в Windows 2000 RAS отобразит ход вызова;
- *RASEO\_SpecificIpAddr* — RAS будет использовать IP-адрес, определенный в поле *ipaddr*;
- III *RASEO\_SpecificNameServers* — RAS будет использовать информацию IP, определенную в полях *ipaddrDns*, *ipaddrDnsAlt*, *ipaddrWins*, и *ipaddrWinsAlt*;
- III *RASEOjSwCompression* — позволяет RAS согласовывать программное сжатие данных, передаваемых через соединение;
- *RASEO\_TerminalAfterDial* — RAS отобразит окно терминала для ввода информации пользователем после установления соединения;
- *RASEO\_TerminalBeforeDial* — RAS отобразит окно терминала для ввода информации пользователем до установления соединения;
- III *RASEOJUseCountryAndAreaCodes* — RAS будет использовать поля *dwCountryID*, *dwCountryCode* и *szAreaCode* для создания телефонного номера с помощью поля *szLocalPhoneNumber*;
- Ж *RASEOJUseLogonCredentials* — RAS будет использовать имя, пароль и домен пользователя, подключившегося во время вызова, но только если включен флаг *RASEO\_RequireMsEncryptedPw*.

***dwCountryID*** — определяет TAPI-идентификатор страны, если включен необязательный флаг *RASEOJUseCountryAndAreaCodes*. Информацию об идентификаторе страны можно получить, вызвав функцию *RasGetCountryInfo*.

**III *dwCountryCode*** — определяет код страны, связанный с полем *dwCountryID* в случае, если включен необязательный флаг *RASEOJUseCountryAndAreaCodes*. Если это поле равно 0, используется код страны, связанный с *dwCountryID*.

**III *szAreaCode*** — определяет код зоны, если включен флаг *RASEOJUseCountryAndAreaCodes*.

**III *szLocalPhoneNumber*** — определяет телефонный номер для вызова. Если включен флаг *RASEOJUseCountryAndAreaCodes*, то RAS объединит поля *dwCountryID*, *dwCountryCode* и *szAreaCode* с вашим телефонным номером.

- ***dwAlternateOffset*** — определяет смещение в байтах от начала этой структуры до места, где хранятся дополнительные телефонные номера для записи телефонного справочника. Дополнительные номера хранятся в виде последовательности строк, завершающихся символом /0. Последняя строка в наборе заканчивается двумя последовательными символами /0.

**III *ipaddr*** — определяет IP-адрес для этого соединения, если включен флаг *RASEO\_SpecificIpAdd*.

**III *ipaddrDns*** — определяет IP-адрес DNS-сервера для этого соединения, если включен флаг *RASEO\_SpecificNameServers*.

**III *ipaddrDnsAlt*** — определяет IP-адрес вспомогательного DNS-сервера для этого соединения, если включен флаг *RASEO\_SpecificNameServers*.

**III *ipaddrWins*** — определяет IP-адрес WINS-сервера для этого соединения, если включен флаг *RASEO\_SpecificNameServers*.

**III *ipaddrWinsAlt*** — определяет IP-адрес вспомогательного WINS-сервера для этого соединения, если включен флаг *RASEO\_SpecificNameServers*.

**III *dwFrameSize*** — изменяет размер кадра протокола на 1006 или 1500 байт, если в поле *dwFramingProtocol* задан флаг *RASFP\_Slip*.

**III *dwfNetProtocols*** — определяет флаги, идентифицирующие, какие сетевые протоколы будут использоваться поверх кадрирующего: *RASNP\_NetBEUI* — для NetBEUI, *RASNPJpx* — для IPX, для *RASNPJp* — для IP.

- ***dwFramingProtocol*** — определяет флаги, идентифицирующие, какой кадрирующий протокол будет использоваться для RAS-соединения: *RASFPJpp* — для PPP, *RASFPSlip* — для SLIP, *RASFP\_Ras* — для асинхронного NetBEUI.
- ***szScript*** — определяет полный путь к сценарию удаленного подключения, который выполняется при установке соединения.

**И *szAutodialDll*** — определяет настраиваемую DLL, которая может быть использована для автоматического вызова (с автонабором номера).

**III *zAutodialFunc*** — определяет имя функции, экспортируемое из запрошенной DLL в поле *szAutodialDll*.

**III *szDeviceType*** — определяет тип устройства, используемого для установки соединения. Значение должно идентифицироваться как строка:

- «RASDT\_Modem» — модем на COM-порте;

- Ж «RASDTJsdn» — адаптер ISDN; » »
  - III «RASDT\_X25» — плата X.25;
  - Я «RASDT\_Vpn» — соединение через виртуальную частную сеть (VPN); \*
  - Ж «RASDT\_Pad» — сборщик (разборщик) пакетов; κ III
  - III «RASDT\_Generic» — базовый тип;
  - «RASDT\_Serial» — последовательный порт; . е»
  - «RASDT\_FrameRelay» — устройство ретрансляции кадров; ц<sub>u</sub> ^.
  - И «RASDT\_Atm» — устройство ATM; »
  - «RASDT\_Sonet» — устройство синхронной оптической сети (SONET);'''
  - III «RASDT\_SW56» — коммутируемый доступ на скорости 56 кбит/с;
  - «RASDTIrd» — ИК-устройство;
  - И «RASDT\_Parallel» — параллельный порт.
- И *szDeviceName* — идентифицирует устройство TAPI, используемое для соединения. Вы можете запрашивать информацию об этих устройствах, используя функцию *RasEnumDevices*.
- III *szX25PadType* — определяет тип X.25 PAD.
- *szX25Address* — определяет адрес X.25.
  - *szX25Facilities* — определяет средства для запроса с узла X.25.
- III *szX25UserData* — определяет дополнительную информацию для соединения X.25.
- *dwChannels* — не используется.
  - *dwReserved1* — не используется, должен иметь значение 0.
  - *dwReserved2* — не используется, должен иметь значение 0.
- III *dwSubEntries* — показывает, сколько многоканальных подзаписей ассоциируется с данной записью телефонного справочника. Нужно присвоить этому полю значение 0 и передать организацию многоканальных подзаписей для него функции *RasSetSubEntryProperties* (подробно будет описана далее в этой главе).
- *dwDialMode* — в Windows 2000 определяет, как RAS должна вызывать многоканальные подзаписи, когда соединение происходит с помощью *RASEDMjDialAll* и *RASEDMJDialAsNeeded*. Если это поле равно *RASEDMjDialAll*, то RAS вызывает все многоканальные подзаписи, а если *RASEDMJDialAsNeeded* — использует поля *dwDialExtraPercent*, *dwDialExtraSampleSeconds*, *dwHangUpExtraPercent* и *dwHangUpExtraSampleSeconds* для определения, когда дополнительные многоканальные подзаписи следует вызывать и отключать.
  - *dwDialExtraPercent* — в Windows 2000 определяет процент общей текущей пропускной способности соединения. RAS вызывает дополнительную подзапись, когда суммарная занятая пропускная способность превышает это значение, минимум на *dwDialExtraSampleSeconds*.

***dwDialExtraSampleSeconds*** — в Windows 2000 определяет количество секунд, на которое может быть превышено процентное использование пропускной способности согласно *dwDialExtraPercent* до того, как будет вызвана дополнительная подзапись.

***dwHangUpExtraPercent*** — в Windows 2000 определяет процентное отношение трафика доступных подзаписей от общей пропускной способности. RAS завершит соединения, связанные с подзаписями, когда данный показатель станет ниже этого значения на время *dwHangUpExtraSampleSeconds*.

• ***dwHangUpExtraSampleSeconds*** — в Windows 2000 определяет количество секунд, на которое должно упасть процентное использование пропускной способности согласно *dwHangUpExtraPercent* до того, как будет отключена дополнительная подзапись.

**III *dwIdleDisconnectSeconds*** — определяет, сколько секунд разрешается простаивать соединению до его завершения. Также вы можете присвоить этому полю *RASIDS\_Disabled*, чтобы предотвратить разрыв соединения, и *RASIDSJseGlobaWalue*, чтобы использовать системное значение по умолчанию.

**III *dwType*** — в Windows 2000 определяет тип записи телефонного справочника:

**III *RASET\_Direct*** — прямое последовательное или параллельное соединение.

- *RASETInternet* — службы соединений с Интернетом (Internet connection services, ICS);
- *RASETPhone* — телефонная линия;

**III *RASETVPN*** — виртуальная частная сеть.

**III *dwEncryptionType*** — в Windows 2000 определяет тип шифрования данных, передаваемых через соединение:

- *ET\_40Bit* — 40-битное шифрование данных;
- *ET\_128Bit* — 128-битное шифрование данных.

• ***dwCustomAuthKey*** — в Windows 2000 определяет ключ аутентификации, предоставляемый поставщику EAP.

**II *guidId*** — в Windows 2000 идентифицирует глобально-уникальный идентификатор GUID, ассоциируемый с записью телефонного справочника.

• ***szCustomDialDll*** — в Windows 2000 определяет путь к DLL, содержащей специальные функции устройства вызова RAS. Если это поле равно *NULL*, то RAS будет использовать стандартное устройство вызова. На подробностях разработки собственного устройства вызова RAS мы останавливаться не будем.

• ***dwVpnStrategy*** — в Windows 2000 определяет стратегию вызова VPN:

**III *VSJDefault*** — сначала RAS вызывает протокол Point-to-Point Tunneling Protocol (PPTP), а если PPTP дает сбой — протокол Layer 2 Tunneling Protocol (L2TP);

III *VS\_L2tpFirst* — сначала RAS вызывает L2TP;

-Л »

- *VS\_L2tpOnly* — RAS вызывает только L2TP;

"

III *VS\_PptpFirst* — сначала RAS вызывает PPTP;

- *VS\_PptpOnly* — RAS вызывает только PPTP;

Когда вы вызываете любую API-функцию RAS, в которой файл телефонного справочника является параметром (*lpszPhonebook*), то можете задать путь к файлу телефонного справочника. Как уже упоминалось, в Windows 9x и CE этот параметр должен иметь значение *NULL*, так как записи телефонного справочника хранятся в системном реестре. В Windows 2000 и NT этот параметр может содержать путь к файлу телефонного справочника. В общем случае файл телефонного справочника будет иметь расширение .pbk, а имя стандартного системного справочника — %SystemRoot%\System32\Ras\ Ras-Phone.pbk. Если указать значение *NULL*, то используется системный телефонный справочник.

Для создания и организации записей телефонного справочника предусмотрены три вспомогательных функции: *RasValidateEntryName*, *RasEnumDevices* и *RasGetCountryInfo*. Функция *RasValidateEntryName* определяет, правильно ли имя отформатировано и существует ли оно уже в телефонном справочнике:

```
DWORD RasValidateEntryName(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

Параметр *lpszPhonebook* — указатель на имя файла телефонного справочника. Параметр *lpszEntry* — строка, представляющая собой имя проверяемой записи. Если имени в справочнике нет и оно отформатировано должным образом, эта функция возвращает *ERROR\_SUCCESS*. Иначе функция дает сбой: с ошибкой *ERROR\_INVALIDJSFAME* — если имя правильно не отформатировано, и с ошибкой *ERROR\_ALREADY\_EXISTS* — если имя уже содержится в справочнике.

Функция *RasEnumDevices* получает имя и тип всех совместимых с RAS устройств, доступных на компьютере:

```
DWORD RasEnumDevices(
    LPRASDEVINFO lpRasDevInfo,
    LPDWORD lpcb,
    LPDWORD lpcDevices
);
```

Параметр *lpRasDevInfo* — указатель на **буфер приложения, который нужен** для получения массива структур *RASDEVINFO*:

```
typedef struct tagRASDEVINFO {
    DWORD dwSize;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASDEVINFO;
```

Поля этой структуры определяются следующим образом:

- **dwSize** — размер (в байтах) структуры *RASDEVINFO* до вызова *RasEnumDevices*;

**III szDeviceType** — строка, описывающая тип устройства, например «RASDT\_Modem»;

**III szDeviceName** — формальное имя TAPI-устройства.

Очень важно, чтобы буфер был достаточно большим для записи нескольких структур, иначе *RasEnumDevices* вернет ошибку *ERROR\_BUFFER\_TOO\_SMALL*. Следующий параметр — *Ipch*, указатель на переменную, получающую количество байт, требующихся для перечисления устройств. Ему нужно присвоить размер (в байтах) вашего *ipRasDevInfo* буфера. Последний параметр — *ipcDevices*, указатель на переменную, получающую число структур *RASDEVINFO*, записанных в *IpRasDevInfo*.

Функция *RasGetCountryInfo* позволяет получать из Windows специальную информацию о стране для TAPI-вызова:

```
DWORD RasGetCountryInfo(
    LPRASCTRYINFO lpRasCtryInfo,
    LPDWORD lpdwSize
);
```

Параметр *lpRasCtryInfo* — буфер, получающий префикс телефонного номера и другую информацию, ассоциируемую с определенной страной. Этот буфер должен содержать структуру *RASCTRYINFO*, за которой следуют дополнительные байты, получающие строку с описанием страны. Для хранения структуры *RASCTRYINFO* и строки описания рекомендуется выделить минимум 256-байтный буфер. Структура *RASCTRYINFO* определена так:

```
typedef struct RASCTRYINFO
{
    DWORD dwSize;
    DWORD dwCountryID;
    DWORD dwNextCountryID;
    DWORD dwCountryCode;
    DWORD dwCountryNameOffset;
} RASCTRYINFO;
```

Ее поля описаны следующим образом:

**III dwSize** — размер (в байтах) структуры *RASCTRYINFO*;

**III dwCountryID** — TAPI-идентификатор страны (относящийся к полю *dwCountryID* структуры *RASENTRY*) в определенном в Windows перечне стран; если его значение равно 1, то будет использована первая запись перечня, и т. д.;

- **dwNextCountryID** — следующий TAPI-идентификатор страны в списке; если это поле получает значение 0, то вы уже в конце списка;
- **dwCountryCode** — префикс телефонного номера представляет собой код, ассоциируемый со страной, определенной в параметре *dwCountryID*,

**III *dwCountryNameOffset*** — сколько байт от начала данной структуры до начала следующей, завершающейся символом /0 строки с описанием страны

Другой параметр *RasGetCountryInfo* — *ipdwSize*, указывает на переменную, получающую количество байт, которое *RasGetCountryInfo* помещает в буфер *ipRasCtryInfo*. Перед вызовом функции присвойте этому параметру значение, определяющее размер буфера приложения

## Добавление записей в телефонный справочник

Программно организовать структуры *RASENTRY* телефонного справочника позволяют четыре функции *RasSetEntryProperties*, *RasGetEntryProperties*, *RasRenameEntry* и *RasDeleteEntry*. Для создания новой или модификации существующей записи используется функция *RasSetEntryProperties*

```
DWORD RasSetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    DWORD dwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    DWORD dwDeviceInfoSize
);
```

Параметр *lpszPhonebook* — указатель на имя файла телефонного справочника. Параметр *lpszEntry* — указатель на строку, используемую для идентификации существующей или новой записи. Если структура *RASENTRY* с таким именем существует, свойства модифицируются, если нет — в телефонном справочнике создается новая запись. Параметр *lpRasEntry* указывает на структуру *RASENTRY*. Для определения дополнительных телефонных номеров вы можете поместить после структуры *RASENTRY* список строк, завершающихся символом /0. Последняя строка завершается двумя последовательными символами /O. Параметр *dwEntryInfoSize* содержит размер (в байтах) структуры, указанной в параметре *lpRasEntry*. Параметр *lpbDeviceInfo* — указатель на буфер с информацией о конфигурации TAPI-устройства. В Windows 2000 и NT этот параметр не используется и должен иметь значение *NULL*. Последний параметр — *dwDeviceInfoSize*, содержит размер (в байтах) буфера *lpbDeviceInfo*.

Функция *RasGetEntryProperties* может быть использована для получения свойств существующей записи телефонного справочника или значений по умолчанию новой

```
DWORD RasGetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    LPDWORD lpdwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    LPDWORD lpdwDeviceInfoSize
);
```

Параметр *ipaszPhonebook* указывает имя файла телефонного справочника, параметр *ipaszEntry* — на строку, идентифицирующую существующую запись справочника. Если присвоить этому параметру значение *NULL*, то параметры *IpRasEntry* и *ipbDeviceInfo* получат стандартные значения записи телефонного справочника. Это полезно, когда будет нужно создать новую запись, вы сможете заполнить поля *IpRasEntry* и *IpbDeviceInfo* правильной информацией о системе перед вызовом функции *RasSetEntryProperties*.

Параметр *IpRasEntry* — указатель на буфер, предоставляемый приложением для получения структуры *RASENTRY*. Как мы уже упоминали, за этой структурой может следовать массив строк, завершающихся символом */0*. Он определяет дополнительные телефонные номера для запрошенной записи телефонного справочника. Следовательно, размер получающего буфера должен быть больше структуры *RASENTRY*. Если передать в этот параметр указатель *NULL*, то *ipdwEntryInfoSize* получит общее количество байт, необходимых для хранения всех элементов структуры *RASENTRY* и дополнительных телефонных номеров.

Параметр *ipdwEntryInfoSize* — указатель на *DWORD*, содержащий количество байт в получающем буфере, который приложение определило через параметр *IpRasEntry*. Когда эта функция выполнится, она обновит *IpdwEntryInfoSize* и запишет в него количество байт, реально полученных в *IpRasEntry*. Мы настоятельно рекомендуем вызывать эту функцию со значением параметра *IpRasEntry*, равным *NULL*, и значением параметра *IpdwEntryInfoSize*, равным 0, для получения информации о размере буфера. Затем вы сможете вызвать эту функцию снова и вывести всю информацию без ошибок.

Параметр *IpbDeviceInfo* указывает на предоставленный приложением буфер, получающий специальную информацию о TAPI-устройстве для данной записи телефонного справочника. Если этот параметр равен *NULL*, то *IpdwDeviceInfoSize* получит число байт, необходимых для получения нужной информации. Если вы используете Windows 2000 и NT, то параметр *IpbDeviceInfo* должен иметь значение *NULL*. Последний параметр — *ipdwDeviceInfoSize*, указатель на *DWORD*, который должен иметь значение количества байт, содержащихся в буфере для *IpbDeviceInfo*. При выполнении *RasGetEntryProperties* *ipdwDeviceInfoSize* возвратит количество байт, которые заносятся в буфер *IpbDeviceInfo*.

Листинг 16-3 демонстрирует, как приложение должно использовать функции *RasGetEntryProperties* и *RasSetEntryProperties* для создания новой записи телефонного справочника.

Листинг 16-3. Создание новой записи телефонного справочника RAS с использованием свойств, заданных по умолчанию

```
<include <windows.h>
<include <ras.h>
<include <raserror.h>
<include <stdio.h>
```

```
void main(void)
```

см. след. стр.



Листинг 16-3.    *{продолжение}*

```

{
    DWORD EntryInfoSize = 0;
    DWORD DeviceInfoSize = 0;
    DWORD Ret;
    LPRASENTRY lpRasEntry;
    LPBYTE lpDeviceInfo;

    // Получение информации о размере буфера

    // для стандартной записи телефонного справочника

    if ((Ret = RasGetEntryProperties(NULL, "", NULL,
        &EntryInfoSize, NULL, &DeviceInfoSize)) != 0)
    {
        if (Ret != ERROR_BUFFER_TOO_SMALL)
        {
            printf("RasGetEntryProperties sizing failed
                "with error Xd\n", Ret);
            return;
        }

        lpRasEntry = (LPRASENTRY) GlobalAlloc(GPTR, EntryInfoSize);

        if (DeviceInfoSize == 0)
            lpDeviceInfo = NULL;
        else
            lpDeviceInfo = (LPBYTE) GlobalAlloc(GPTR, DeviceInfoSize);

        // Получение стандартной записи телефонного справочника

        lpRasEntry->dwSize = sizeof(RASENTRY);

        if ((Ret = RasGetEntryProperties(NULL, "", lpRasEntry,
            &EntryInfoSize, lpDeviceInfo, &DeviceInfoSize)) != 0)
        {
            printf("RasGetEntryProperties failed with error Xd\n",
                Ret);
            return;
        }

        // Подтверждение нового имени телефонного справочника "Testentry"
        if ((Ret = RasValidateEntryName(NULL, "Testentry")) !=
            ERROR_SUCCESS)
        {
            printf("RasValidateEntryName failed with error Xd\n",
                Ret);
            return;
        }
    }
}

```

**Листинг 16-3.** (продолжение)

```

// Установка новой записи телефонного справочника "Testentry",
// используя стандартные свойства

if ((Ret = RasSetEntryProperties(NULL, "Testentry",
    lpRasEntry, EntryInfoSize, lpDeviceInfo,
    DeviceInfoSize)) != 0)

    printfC'RasSetEntryProperties failed with error %d\n",
a      Ret);
    return;
}
}

```

**Переименование записи телефонного справочника**

Теперь рассмотрим функцию *RasRenameEntry*, которая позволяет переименовать записи телефонного справочника:

```

DWORD RasRenameEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszOldEntry,
    LPCTSTR lpszNewEntry

```

Параметр *lpszPhonebook* указывает на имя файла телефонного справочника, а *lpszOldEntry* — на строку, идентифицирующую существующую запись, которую вы намерены переименовать. Параметр *lpszNewEntry* — указатель на строку, содержащую новое имя для записи телефонного справочника. Приложение должно вызвать функцию *RasValidateEntryName* для нового имени перед вызовом *RasRenameEntry*. Если *RasRenameEntry* выполняется успешно, она возвращает 0. Если же происходит сбой, она возвращает следующие типы ошибок:

- **ERRORINVAHDNAME** — имя, переданное в *lpszNewEntry*, неверно;
- III ERRORALREADYEXISTS** — имя, переданное в *lpszNewEntry*, уже есть в телефонном справочнике;
- III ERROR CANNOT FIND PHONEBOOKENTRY** — в телефонном справочнике не найдено имя, указанное в *lpszOldEntry*.

**Удаление записей из телефонного справочника**

Чтобы удалить записи из телефонного справочника, вызовите функцию *RasDeleteEntry*—.

```

DWORD RasDeleteEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry

```



вает на переменную, которая получает количество структур *RASENTRYNAA1E* записываемых в буфер *lprasentryname*.

## Управление реквизитами пользователя

Когда клиент RAS устанавливает соединение, используя запись телефо.шного справочника через *RasDial*, он сохраняет реквизиты безопасности пользователей и ассоциирует их с записями телефонного справочника. Функции *RasGetCredentials*, *RasSetCredentials*, *RasGetEntryDialParams* и *RasSetEntryDialParams* позволяют организовать реквизиты безопасности пользователей, ассоциируемых с записями телефонного справочника. Функции *RasGetCredentials* и *RasSetCredentials* были внедрены в Windows NT 4, они также доступны в Windows 2000. Эти две функции заменяют *RasGetEntryDialParams* и *RasSetEntryDialParams*. Поскольку *RasGetCredentials* и *RasSetCredentials* не доступны в Windows 9x и CE, вы можете использовать *RasGetEntryDialParams* и *RasSetEntryDialParams* для этой цели на всех платформах.

Функция *RasGetCredentials* выводит реквизиты пользователя, связанные с записью телефонного справочника:

```
DWORD RasGetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials
);
```

Параметр *lpszPhonebook* — указатель на имя файла телефонного справочника. Параметр *lpszEntry* — строка, представляющая существующую запись справочника. Параметр *lpCredentials* указывает на структуру, которая может получать имя пользователя, пароль и домен, ассоциируемые с записью телефонного справочника:

```
typedef struct {
    DWORD dwSize;
    DWORD dwMask;
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
} RASCREDENTIALS, «LPRASCREDENTIALS;
```

Поля этой структуры определяются следующим образом:

- М *dwSize* — всегда: размер (в байтах) структуры *RASCREDENTIALS*;
- III *dwMask* — поле битовой маски идентифицирует соответствие: флага *RASCMJUserName* — полю *szUserName*, флага *RASCMJ\_Password* — полю *szPassword*, флага *RASCMJDomain* — полю *szDomain*.
- III *szUserName* — содержит имя пользователя для входа в систему, завершается /0;
- III *szPassword* — содержит пароль пользователя для входа в систему, завершается /0;

III *szDomain* — содержит домен для входа пользователя в систему, завершается /O

Если *RasGetCredentials* выполняется успешно, то она возвращает 0 Приложение способно определить, какие реквизиты безопасности заданы в соответствии с полем *divMask* структуры *IpCredentials*

Функция *RasSetCredentials* подобна *RasGetCredentials*, за исключением того, что позволяет изменять реквизиты безопасности, связанные с записью телефонного справочника Кроме того, *RasSetCredentials* обладает дополнительным параметром *—/Clear-Credentials*

```
DWORD RasSetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS IpCredentials,
    BOOL fClearCredentials
```

),

Параметр *fClearCredentials* — это логический оператор Если он равен *TRUE*, то функция *RasSetCredentials* заменяет реквизиты, указанные в поле *dwMask* структуры *IpCredentials*, на пустые строки (<») Например, если поле *divMask* содержит флаг *RASCM\_Password*, то заданный пароль заменяется на пустую строку Если функция *RasSetCredentials* выполняется успешно, то она возвращает 0

Также для организации реквизитов безопасности пользователя, связанных с записью телефонного справочника, вы можете использовать *RasGetEntryDialParams* и *RasSetEntryDialParams*

```
DWORD RasGetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    LPBOOL lpfPassword
```

< \*3Д  
, - ^j

Параметр *lpszPhonebook* указывает на имя файла телефонного справочника, а *lprasdialparams* — на структуру *RASDIALPARAMS* Параметр *lpfPassword* это логический флаг, который возвращает *TRUE*, если пароль пользователя был получен в структуре *lprasdialparams*

Функция *RasSetEntryDialParams* изменяет информацию о соединении, которая была задана при последнем вызове *RasDial* для конкретной записи телефонного справочника

```
DWORD RasSetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    BOOL fRemovePassword
```

),

Параметры *lpszPhonebook* и *lprasdialparams* такие же, как и в *RasGetEntryDialParams* Параметр *fRemovePassword* — это логический флаг Если он равен *TRUE*, то *RasSetEntryDialParams* удаляет пароль, ассоциируемый с записью телефонного справочника, которая указана в структуре *lprasdialparams*

## Многоканальные подзаписи телефонного справочника

В Windows 2000 и NT RAS позволяет организовать многоканальные подзаписи телефонного справочника для усовершенствования возможностей соединения. Многоканальные соединения дают возможность использовать более одного устройства для соединения, ассоциируемого с RAS-соединением. Это помогает увеличить общую пропускную способность соединения. Организовывать многоканальные подзаписи можно с помощью функций *RasGetSubEntryProperties* и *RasSetSubEntryProperties*.

```
DWORD RasGetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY lpRasSubEntry,
    LPDWORD lpdwcb,
    LPBYTE lpbDeviceConfig,
    LPDWORD lpcbDeviceConfig
),
```

Параметр *lpszPhonebook* — указатель на имя файла телефонного справочника. Параметр *lpszEntry* — запись справочника, индекс подзаписи из которой определяет параметр *dwSubEntry*. Параметр *lpRasSubEntry* — указатель на буфер, который будет получать структуру *RASSUBENTRY*, за ним следует список дополнительных телефонных номеров. Структура *RASSUBENTRY* определена так:

```
typedef struct tagRASSUBENTRY
{
    DWORD dwSize,
    DWORD dwfFlags,
    TCHAR szDeviceType[RAS_MaxDeviceType + 1],
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD dwAlternateOffset;
} RASSUBENTRY;
```

Поля этой структуры можно описать следующим образом:

- ***dwSize*** — размер (в байтах) структуры *RASSUBENTRY*,

**III *dwFlags*** — не используется,

- ***szDeviceType*** — строка, представляющая тип устройства, используемого для соединения,
- ***szDeviceName*** — действительное имя TAPI-устройства,

**III *szLocalPhoneNumber*** — телефонный номер для использования этим устройством,

- ***dwAlternateOffset*** — количество байт от начала структуры до списка последовательных разделенных нулями строк, которые следуют за структурой

Буфер *ipRasSubEntry* должен быть достаточно большим, чтобы вместить структуру *RASSUBENTRY* и дополнительные телефонные номера. Иначе *RasGetSubEntryProperties* вернет ошибку *ERROR\_BUFFER\_TOO\_SMALL*. Параметру *Ipdwcb* нужно присвоить количество байт в буфере *IpRasSubEntry*. При выполнении *Ipdwcb* будет присвоено общее количество байт, необходимых для хранения структуры и дополнительных номеров. Параметры *ipbDeviceConfig* и *ipcbDeviceConfig* не используются и должны быть равны *NULL*.

Вы можете создать новую или модифицировать существующую подзапись определенной записи телефонного справочника с помощью функции *RasSetSubEntryProperties*.

```
DWORD RasSetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY IpRasSubEntry,
    DWORD dwcbRasSubEntry,
    LPBYTE IpbDeviceConfig,
    DWORD dwcbDeviceConfig
),
```

Параметры ее такие же, как и у *RasGetSubEntryProperties*, за исключением *IpRasSubEntry*, который определяет подзапись для добавления в телефонный справочник.

## Управление соединением

Получить свойства соединения, установленного в вашей системе, позволяют следующие RAS-функции *RasEnumConnections*, *RasGetSubEntryHandle* и *RasGetProjectionInfo*. Функция *RasEnumConnections* перечисляет все активные RAS-соединения.

```
DWORD RasEnumConnections(
    LPRASCONN lprasconn,
    LPDWORD lpcb,
    LPDWORD lpcConnections
),
```

Параметр *lprasconn* — это буфер приложения, который будет получать массив структур *RASCONN*.

```
typedef struct _RASCONN
{
    DWORD dwSize,
    HRASCONN hrasconn,
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#ifdef WINVER
    CHAR szDeviceType[RAS_MaxDeviceType + 1],
    CHAR szDeviceName[RAS_MaxDeviceName + 1];
#endif
} RASCONN;
```

```

    CHAR szPhonebook[MAX_PATH],
    DWORD dwSubEntry,
#endif
#ifdef (WINVER >= 0x500)
    GUID guidEntry,
#endif
} RASCONN,

```

Эти поля определяются следующим образом

- **dwSize** — размер (в байтах) структуры *RASCONN*,
- III **brasconn** — созданный функцией *RasDial* описатель соединения,
- III **szEntryName** — запись телефонного справочника, используемая для установления соединения, если была задействована пустая строка, то поле получит строку с точкой (.), а за ней — используемый телефонный номер,
- III **szDeviceType** — строка описывает тип устройства, используемого при соединении,
- III **szDeviceName** — строка с именем устройства, с помощью которого устанавливалось соединение,
- III **szPhonebook** — полный путь к телефонному справочнику для записи, по которой устанавливалось соединение,
- III **dwSubEntry** — индекс подзаписи записи телефонного справочника,
- **guidEntry** — в Windows 2000 получает GUID для записи телефонного справочника, используемой для установления соединения

Необходимо передать в *RasEnumConnections* достаточно большой буфер для хранения нескольких структур *RASCONN*, иначе функция даст сбой с ошибкой *ERROR\_BUFFER\_TOO\_SMALL*. Следовательно, первая структура *RASCONN* в буфере должна иметь поле *dwSize* со значением размера структуры *RASCONN* в байтах. Следующий параметр — *Ipcb*, указатель на переменную, которой необходимо присвоить размер (в байтах) массива *Iprascconn*. В нем будет содержаться число байт, требуемых для перечисления всех соединений при выполнении функции. Если размер буфера окажется мал, вы всегда можете попытаться повторить эту операцию с увеличенным размером буфера, возвращенном в *Ipcb*. Параметр *ipcConnections* — указатель на переменную, которая получает число записанных в *Iprascconn* структур *RASCONN*.

Функция *RasGetSubEntryHandle* позволяет получать описатель для конкретной подзаписи многоканального соединения

```

DWORD RasGetSubEntryHandle(
    HRASCONN hrasconn,
    DWORD dwSubEntry,
    LPHRASCONN lphrasconn
),

```

Параметр *hrasconn* — описатель RAS-соединения для многоканального соединения, а *dwSubEntry* — индекс подзаписи устройства в этом соединении. Параметр *lphrasconn* получает описатель соединения для устройства подзаписи.



Описатели соединения, получаемые от *RasEnumConnections* и *RasGetSubEntryHandle*, предоставляют информацию о сетевом протоколе, используемом в соединении. Такая информация называется *проекционной*. Сервер удаленного доступа использует ее для представления удаленного клиента в сети. Например, при установлении RAS-соединения, использующего протокол IP над кадрным протоколом, информация о конфигурации IP (такая, как назначенный IP-адрес) определяется с помощью службы RAS клиента. Вы можете вывести проекционную информацию для протоколов, которые передаются через кадрный протокол PPP при вызове функции *RasGetProjectionInfo*:

```
DWORD RasGetProjectionInfo(                                     gg
    HRASCONN hrasconn,
    RASPROJECTION rasprojection,
    LPVOID lpprojection,
    LPDWORD lpcb
);
```

Параметр *hrasconn* — описатель соединения RAS. Параметр *rasprojection* — перечислимый тип *RASPROJECTION*, позволяющий определять, для какого протокола получать информацию. Параметр *lpprojection* получает структуру данных, которая ассоциируется с перечислимым типом, указанным в *rasprojection*. Последний параметр — *lpcb*, указатель на переменную, которой необходимо присвоить размер структуры *lpprojection*. После выполнения функции эта переменная будет содержать размер буфера для получения проекционной информации.

Получать информацию о соединении позволяют следующие перечислимые типы *RASPROJECTION*: *RASP\_Amb*, *RASP\_PppNbf*, *RASP\_PppIp* и *RASPPpIp*. Если вы укажете перечислимый тип *RASP\_Amb*, то получите структуру *RASAMB*.

```
typedef struct _RASAMB
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASAMB;
```

Поля определены следующим образом:

- III ***dwSize*** — размер (в байтах) структуры *RASAMB*; 1WT
- ***dwError*** — код ошибки от процесса согласования PPP;
- III ***szNetBiosError*** — NetBIOS-имя, если во время процесса аутентификации PPP возникает конфликт имен; если *dwError* возвращает *ERROR\_NAME\_EXISTS* *JDN\_NET*, то *szNetBiosError* получит имя, вызвавшее ошибку;
- ***bLana*** — идентифицирует NetBIOS-номер LANA, который использовался для установления соединения удаленного доступа.

Если вы укажете перечислимый тип *RASP\_PppNbf* & *RasGetProjectionInfo*, то получите структуру *RASPPPNBF*.

```
typedef struct _RASPPPNBF      f!
<
    DWORD dwSize;
    .  DWORD dwError;
    DWORD dwNetBiosError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    TCHAR szWorkstationName[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASPPPNBF;
```

Поля структуры *RASPPPNBF* аналогичны полям структуры *RASAMB*, за исключением двух дополнительных: *szWorkstationName* и *dwNetBiosError*. Поле *szWorkstationName* получает имя NetBIOS, которое было использовано для идентификации рабочей станции в сети, к которой вы подключаетесь. В поле *dwNetBiosError* записывается произошедшая ошибка NetBIOS.

Если определить тип перечисления *RASP\_Ppplx*, как *RasGetProjectionInfo*, то вы получите структуру *RASPPPIX*:

```
typedef struct _RASPPPIX
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpAddress[RAS_MaxIpAddress + 1];
} RASPPPIX;
```

Поля определены следующим образом:

- *dwSize* — размер (в байтах) структуры *RASPPPIX*;

*III dwError* — код ошибки от процесса согласования PPP;

*III szlpxAddress* — строка, представляющая IPX-адрес клиента в удаленной сети.

Если вы определите тип перечисления *RASPppIp*, как *RasGetProjectionInfo*, то получите структуру *RASPPPIP*:

```
typedef struct _RASPPPIP
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpAdress[RAS_MaxIpAddress + 1];
    TCHAR szServerIpAdress[RAS_MaxIpAddress + 1];
} RASPPPIP;
```

Поля определены следующим образом:

- *dwSize* — размер (в байтах) структуры *RASPPPIP*;

*III dwError* — код ошибки от процесса согласования PPP;

*III szIpAdress* — строка, представляющая IP-адрес клиента в удаленной сети;

*Я szServerIpAdress* — строка, представляющая IP-адрес сервера.

Листинг 16-4 демонстрирует, как выводить IP-адреса, заданные для IP-соединения, устанавливаемого через RAS.

**Листинг 16-4. Использование *RasGetProjectionInfo* при IP-соединении**

```

IpProjection = (RASPPPIP *) GlobalAlloc(GPTR, cb);
IpProjection->dwSize = sizeof(RASPPPIP);
cb = sizeof(RASPPPIP);

Ret = RasGetProjectionInfo(hRasConn, RASP_PppIp,
    IpProjection, &cb);

if (Ret != ERROR_SUCCESS)
{
    printf("RasGetProjectionInfo failed with error Xd", Ret);
    return;
}

else

    printf("\nRas Client IP address: Xs\n",
        IpProjection->szIpAddress);
    printf("Ras Server IP address: Xs\n",
        IpProjection->szServerIpAddress);

```

## Резюме

В этой главе, завершающей часть III и всю книгу, были рассмотрены основы использования RAS для расширения сетевых возможностей компьютера. Мы описали, как вызывается функция *RasDial* для взаимодействия с удаленными компьютерами, и то, как далее использовать возможности RAS посредством создания записей телефонного справочника.

КО! П Р И Л О Ж Е Н И Е

Перечень команд NetBIOS

т, »W Ъ

В этом приложении перечислены и описаны допустимые команды для поля *ncbcommand* из структуры *NCB*, которую следует передать функции *Netbios*. В описание каждой команды входит таблица, где указано, какие поля вы должны задать для этой команды и какие поля функция *Netbios* задает перед возвратом. В каждой таблице по две колонки. Первая показывает, является ли данное имя из структуры *NCB* параметром ввода или вывода. Во второй колонке сообщается, должно ли поле задаваться при вызове NetBIOS. Подробное описание функции *Netbios* см в главе 1.

Команда **NCBADDGRNAME**

Команда добавляет имя группы в локальную таблицу имен. Это имя не должно конфликтовать с уникальным, но кто-то другой может использовать его в качестве имени группы. Имена групп в основном применяют как приемники дейтаграмм. Номер имени возвращается в поле *ncbnum*, используемое в операциях с дейтаграммами.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>ncbretcode</i>	Вывод	Нет
<i>neb Isn</i>	Не обязательно	Нет
<i>ncb_num</i>	Вывод	Нет
<i>ncb_buffer</i>	Не обязательно	Нет
<i>neb length</i> (зН	Не обязательно	Нет
<i>neb callname</i> «Д	Не обязательно	Нет
<i>neb name</i> еД	Ввод	Да
<i>neb rto</i> *Д	Не обязательно	Нет
<i>ncb_sto</i> тЭй	Не обязательно	Нет
<i>ncb_post</i> *	Ввод	Нет
<i>neb lana num.</i> -л	Ввод	Да
<i>neb cmd cplt</i>	Вывод	Нет
<i>neb event</i>	Ввод	Нет

Команда **NCBADDNAME**

Команда добавляет уникальное имя в локальную таблицу имен. Если имя в сети не уникально, программа выдаст ошибку. Номер имени возвращается в поле *ncbjnum*, используемом в операциях с дейтаграммами.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>ncbretcode</i>	Вывод	Нет
<i>ncbjns</i>	Не обязательно	Нет
<i>ncbjium</i>	Вывод	Нет
<i>ncbbuffer</i>	Не обязательно	Нет
<i>ncbjength</i>	Не обязательно	Нет
<i>neb callname</i>	Не обязательно	Нет
<i>ncb_name</i>	Ввод	Да
<i>ncbjrto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>nebjanaajnum</i>	Ввод	Да
<i>neb cmd cplt</i>	Вывод	Нет
<i>ncbevent</i>	Ввод	Нет

Команда **NCBASTAT**

Команда возвращает сведения о состоянии локального или удаленного адаптера. При ее вызове поле *ncb\_buffer* должно ссылаться на буфер со структурой *ADAPTER STATUS*, за которой следует массив структур *NAMEJ3UFFER*.

Поле	Ввод или вывод	Нужно ли задать
<i>ncbcommand</i>	Ввод	Да
<i>nebjretcode</i>	Вывод	Нет
<i>ncbjns</i>	Не обязательно	Нет
<i>nebjium</i>	Необязательно	Нет \ ,
<i>nebjbuffer</i>	Ввод и вывод	Да
<i>ncbjength</i>	Ввод и вывод	Да
<i>ncbcallname</i>	Ввод	Да
<i>nebjname</i>	Не обязательно	Нет
<i>ncbrto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>nebjananum</i>	Ввод	Да
<i>ncbcmdcplt</i>	Вывод	Нет
<i>ncb_event</i>	Ввод	Нет

## Команда **NCBCALL**

Команда устанавливает сетевое соединение с другим процессом, указанным в поле *neb name*.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>neb Isn</i>	Вывод	Нет
<i>ncbjum</i>	Не обязательно	Нет <sup>1</sup> <sub>т ч</sub> tyP
<i>neb buffer</i>	Не обязательно	Нет ,,, ", "
<i>neb length</i>	Не обязательно	Нет ч
<i>ncbcallname</i>	Ввод	Да
<i>neb name</i>	Ввод \ (лит	Да " * "" • "">
<i>ncbrto</i>	Ввод	Нет
<i>neb sto</i>	Ввод	Нет w
<i>ncb_post</i>	Ввод	Нет
<i>neb lana num</i>	Ввод	Да <
<i>neb cmd cplt</i>	Вывод	Нет <
<i>neb event</i>	Ввод	Нет

## Команда **NCBCANCEL**

Эта команда отменяет предыдущую. Поле *ncb\_buffer* указывает на структуру *NCB* с операцией, которую вы хотите отменить. Отмена уведомляющих команд *NCBSEND* или *NCBCHAINSEND* разрывает сеанс, отмена их не уведомляющих вариантов — нет. Нельзя отменить команды *NCBADDGRNAME*, *NCBADDNAME*, *NCBCANCEL*, *NCBDELNAME*, *NCBRESET*, *NCBDGSEND*, *NCBDGSENDBC* и *NCBSSTAT*.

Поле	Ввод или вывод	Нужно ли задать
<i>nebcommand</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Не обязательно	Нет
<i>neb num.</i>	Не обязательно	Нет
<i>nebbuffer</i>	Ввод	Да
<i>neb length</i>	Не обязательно	Нет
<i>neb callname</i>	Не обязательно	Нет
<i>nebname</i>	Не обязательно	Нет
<i>ncbrto</i>	Не обязательно	Нет
<i>ncbsto</i>	Не обязательно	Нет
<i>ncb_post</i>	Не обязательно	Нет
<i>neb lana_num</i>	Ввод	Да
<i>neb cmd cplt</i>	Вывод	Нет
<i>neb event</i>	Не обязательно	Нет

## Команда NCBCHAINSENDX1ЮКЖбднвмЖЖ

Команда отправляет содержимое двух буферов указанному получателю. Максимальное количество данных, которое можно отправить, — 128 кб (по 64 кб в каждом буфере). Укажите первый буфер и его длину в полях *neb buffer* и *ncbjength* соответственно. Используйте байты 0-1 из *ncbcallname*, чтобы задать длину второго буфера, а байты 2-5 — чтобы сослаться на него.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>neb num</i>	Необязательно	Нет
<i>ncbj&gt;uffer</i>	Ввод	Да
<i>neb length</i>	Ввод	Да
<i>neb callname</i> IMl	Ввод	Да
<i>neb name</i> nil	Необязательно	Нет
<i>ncbrto</i>	Необязательно	Нет
<i>nebsto</i>	Необязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>neb lana num</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод »;	Нет
<i>neb event</i>	Ввод	Нет

## КомандаNCBCHAINSENDNA

Команда отправляет содержимое двух буферов указанному получателю, не ожидая подтверждения. Максимальное количество данных, которое можно отправить, — 128 кб (по 64 кб в каждом буфере). Укажите первый буфер и его длину в полях *nebjbuffer* и *ncbjlength* соответственно. Используйте байты 0-1 из *ncbjoallname*, чтобы задать длину второго буфера, а байты 2-5 — чтобы сослаться на него.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Да
<i>ncbretcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>neb num</i>	Необязательно	Нет
<i>neb buffer</i>	Ввод	Да
<i>ncbjength</i>	Ввод	Да
<i>ncbjzallname</i>	Ввод	Да
<i>nebjame</i>	Необязательно	Нет
<i>ncb_rto</i>	Необязательно	Нет
<i>ncb_sto</i>	Необязательно	Нет
<i>neb post</i>	Ввод	Нет

продолжение

Поле	Ввод или вывод	Нужно ли задать
<i>neb lana num</i>	Ввод	Да
<i>ncbcmcdpl</i>	Вывод	Нет
<i>neb event</i>	Ввод	Нет

## Команда **NCBDELNAME**

Команда удаляет имя из локальной таблицы имен. Если удаляемое имя связано с активными сеансами, возвращается ошибка *NRC^ACTSES* (0x0F). Если ожидают выполнения какие-либо неактивные сеансы — ошибка *NRC\_NAMERR* (0x17).

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>ncb_retcode</i>	Вывод	Нет
<i>neb Isn</i>	Не обязательно	Нет
<i>nebjium</i>	Не обязательно	Нет
<i>neb buffer</i>	Не обязательно	Нет
<i>neb length</i>	Не обязательно	Нет
<i>neb callname</i>	Не обязательно	Нет
<i>neb name</i>	Ввод	Да
<i>neb rto</i>	Не обязательно	Нет
<i>neb sto</i>	Не обязательно	Нет
<i>nebijost</i>	Ввод	Нет
<i>neb lana num</i>	Ввод	Да
<i>neb cmdcpl</i>	Вывод	Нет
<i>ncb_event</i>	Ввод	Нет

## Команда **NCBDGRECV**

Команда получает дейтаграмму, адресованную локальному имени, связанному со значением *nebjium*. Если *ncbjium* — 0xFF, то команда получает дейтаграммы, адресованные любому локальному имени. Локальное имя может быть именем группы, либо уникальным. Если при отправке дейтаграммы нет команд, ожидающих приема дейтаграмм, данные удаляются. Если предоставленный буфер слишком мал, появится сообщение о невозможности завершения (*NRCJNCOMP* — 0x06) и данные будут усечены до размера буфера.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>neb Isn</i>	Не обязательно	Нет
<i>neb num</i>	Ввод	Да
<i>nebbuffer</i>	Вывод	Да

см. след. стр.



продолжение

Поле	Ввод или вывод	Нужно ли задать	
<i>ncbjlength</i>	Ввод и вывод	Да	
<i>neb callname</i>	Вывод	Нет	
<i>neb name</i>	Не обязательно	Нет	1
<i>neb rto</i>	Не обязательно	Нет	
<i>neb sto</i>	Не обязательно	Нет	1
<i>ncb_post</i>	Ввод	Нет	
<i>nebjanajvum</i>	Ввод	Да	1
<i>neb cmd cplt</i>	Вывод	Нет	11
<i>ncb_event</i>	Ввод	Нет	

## Команда **NCBDGRCVBC**

Команда получает широковещательную дейтаграмму от любого имени, выдающего команды отправки таких дейтаграмм. Если предоставленный буфер слишком мал, появится сообщение о невозможности завершения (*NRCJN-COMP* — 0x06) и данные будут усечены до размера буфера.

Поле	Ввод или вывод	Нужно ли задать	
<i>nebcommand</i>	Ввод	Да	
<i>ncbretcode</i>	Вывод	Нет	н
<i>ncbjsn</i>	Не обязательно	Нет	<
<i>nebjium</i>	Ввод	Да	.1
<i>ncb_buffer</i>	Ввод	Да	л
<i>ncbjlength</i>	Ввод и вывод	Да	\.
<i>neb callname</i>	Вывод	Нет	
<i>nebjame</i>	Не обязательно	Нет	si
<i>ncb_rto</i>	Не обязательно	Нет	
<i>ncb_sto</i>	Необязательно	Нет	\ •
<i>ncb_post</i>	Ввод	Нет	
<i>nebjanajnum</i>	Ввод	Да	fr
<i>ncbcmdcplt</i>	Вывод	Нет	"
<i>ncbevent</i>	Ввод	Нет	

## Команда **NCBDGSEND**

Команда отправляет дейтаграмму по указанному уникальному или групповому имени. Если в адаптере есть отложенная команда приема дейтаграммы для того же самого имени, то адаптер получит свое собственное сообщение. Выполните локальную команду *NCBASTAT* — в структуре *ADAPTERJ5STATUS* будет возвращен максимальный размер дейтаграммы для базового транспортного протокола.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Не обязательно	Нет
<i>neb num</i>	Ввод	Да
<i>neb buffer</i>	Ввод	Да
<i>nebjength</i>	Ввод	Да
<i>neb callname</i>	Ввод	Да
<i>nebjname</i>	Не обязательно	Нет
<i>neb rto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i> ~~*	Ввод	Нет
<i>neb lananum</i>	Ввод	Да
<i>ncbcmcdclt</i>	Вывод	Нет
<i>neb event</i>	Ввод	Нет

## Команда *NCBDGSEND*BC

Команда отправляет широковещательную дейтаграмму каждому узлу в локальной сети. Только компьютеры с ожидающими выполнения командами приема дейтаграмм получают сообщение. Если в локальном адаптере есть отложенная команда приема дейтаграммы, он получит свое собственное сообщение. Широковещательные дейтаграммы имеют ограничения по размеру, упомянутые в описании *NCBDGSEND*.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>neb Isn</i>	Ввод	Да
<i>nebnum</i>	Ввод	Да
<i>neb buffer</i>	Ввод	Да
<i>nebjength</i>	Не обязательно	Нет
<i>neb callname</i>	Не обязательно	Нет
<i>neb name</i>	Не обязательно	Нет
<i>neb rto</i>	Не обязательно	Нет
<i>neb sto</i>	Не обязательно	Нет
<i>ncb_post</i> '	Ввод	Нет
<i>neb lana num</i> 'I	Ввод	Да
<i>neb cmd cplt</i> ^	Вывод	Нет
<i>neb event</i>	Ввод	Нет

## Команда **NCBENUM**

Команда нумерует LANA. При этом *ncbjouffer* должен ссылаться на структуру *LANA\_ENUM*. По возвращении поле *length* структуры *LANA\_ENUM* будет содержать количество номеров LANA на локальном компьютере. Поле *lane* из *LANA\_ENUM* заполняется номерами LANA.

Поле	Ввод или вывод	Нужно ли задать
<i>neb command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>ncbjns</i>	Не обязательно	Нет
<i>neb num</i>	Не обязательно	Нет
<i>ncbjouffer</i>	Ввод	Да
<i>neb length</i>	Ввод	Да
<i>ncbcallname</i>	Не обязательно	Нет
<i>neb name</i>	Не обязательно	Нет
<i>ncbrto</i>	Не обязательно	Нет
<i>neb sto</i>	Не обязательно	Нет
<i>ncbjost</i>	Не обязательно	Нет
<i>neb lane num</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb event</i>	Не обязательно	Нет

## Команда **NCBFINDNAME**

Команда находит местоположение (имя компьютера) в сети. После ее выполнения буфер *ncbjouffer* заполняется структурой *EINDJVAME\_HEADER*, за которой следуют одна или несколько структур *FINDJVAMEJ3UFFER*. Команда специфична для Microsoft Windows NT и не поддерживается другими платформами Win32.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Да
<i>neb retcode</i>	Вывод	Нет
<i>neb Isn</i>	Не обязательно	Нет
<i>nebnum</i>	Не обязательно	Нет
<i>neb buffer</i>	Ввод и вывод	Да
<i>nebjlength</i>	Ввод	Да
<i>ncb_callname</i>	Ввод	Да
<i>neb name</i>	Не обязательно	Нет
<i>neb rto</i>	Не обязательно	Нет
<i>nebsto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>nebjanenum</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb event</i>	Ввод	Нет

# КомандаNCBHANGUP

Команда закрывает указанный сеанс. Все отложенные команды приема для сетевого соединения завершаются и возвращается ошибка закрытия сеанса: *NRC SCLOSED* — ОхОА. Если команды отправки или их цепочка ожидают обработки, то команда завершения сеанса откладывается, пока они не будут выполнены. Эта задержка возникает, если команды передают данные или ожидают, пока удаленная сторона выдаст команду приема. К тому же, если в режиме ожидания находятся несколько команд *NCBRECVDANY*, только одна из них вернет код ошибки, когда сеанс будет завершен. Для любой другой команды приема код ошибки возвращает каждая ожидающая команда

Поле	Ввод или вывод	Нужно ли задать
<i>ncbcommand</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>neb_num</i>	Не обязательно	Нет
<i>nebjouffer</i>	Не обязательно	Нет — — ~
<i>neb_length</i>	Не обязательно	Нет —
<i>ncb_callname</i>	Не обязательно	Нет
<i>neb_name</i>	Не обязательно	Нет
<i>neb_rto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>neb_lana_num</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb_event</i>	Ввод	Нет

# КомандаNCBLANSTALERT

Команда специфична для Windows NT и уведомляет пользователя о сбоях в локальной сети, длящихся больше минуты. Впрочем, при тестировании эта команда не реагировала на ряд типичных ошибок ЛВС, например, на разрыв сетевого кабеля.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Да
<i>nebjretcode</i>	Вывод	Нет
<i>ncbjsn</i>	Не обязательно	Нет
<i>nebjnum</i>	Не обязательно	Нет
<i>ncb_buffer</i>	Не обязательно	Нет
<i>ncbjlength</i>	Не обязательно	Нет
<i>ncb_callname</i>	Не обязательно	Нет
<i>neb_name</i>	Не обязательно	Нет
<i>ncbjrto</i>	Не обязательно	Нет

продолжение

Поле	Ввод или вывод	Нужно ли задать
<i>ncbjto</i>	Не обязательно	Нет
<i>ncb_post</i>	Не обязательно	Нет
<i>neb lananum</i>	Ввод	Да
<i>neb cmd_cplt</i>	Вывод	Нет
<i>neb event</i>		Нет

## Команда **NCBLISTEN**

Команда прослушивает соединения от другого процесса, локального или удаленного. Если первый символ *ncb\_callname* — звездочка (\*), соединение устанавливается с любым адаптером сети, выдавшим команду *NCBCALL* локальному имени. Имя, генерирующее *NCBCALL*, возвращается в поле *neb callname*. Если задан таймаут отправки или приема, он применяется во всех запросах в рамках нового сеанса.

Поле	Ввод или вывод	Нужно ли задать
<i>nebcommand</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Вывод	Нет
<i>neb num</i>	Не обязательно	Нет
<i>ncb_buffer</i>	Не обязательно	Нет
<i>neb length</i>	Не обязательно	Нет
<i>neb_callname</i>	Ввод и вывод	Да
<i>neb name</i>	Ввод	Да
<i>ncbjrto</i>	Ввод	Нет
<i>neb sto</i>	Ввод	Нет
<i>ncb_post</i>	Ввод	Нет
<i>nebjanajnum</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb event</i>	Ввод	Нет

)Л

## Команда **NCBREC V**

Команда получает данные от сеанса с указанным именем. Если ожидающие обработки данные способны получить несколько команд, они обрабатываются в таком порядке.

1. Прием (*NCBREC V*).
2. Прием всех для указанного имени (*NCBREC VANY*).
- 3- Прием всех для любого имени (*NCBREC VANY*).

Все команды с одинаковым приоритетом обрабатываются в порядке поступления. Если буфер обмена не достаточно вместителен, выдается ошиб-

ка *NRCJNCOMP* (0x06). Тогда выдайте еще одну команду приема с более емким буфером. Впрочем, если в команде отправки задан таймаут или она не требует подтверждения, данные будут потеряны. По завершении в поле *ncbjlength* будет храниться объем фактически считанных данных.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>ncb_num</i>	Ввод	Да
<i>neb_buffer</i>	Ввод	Да
<i>neb_length</i>	Ввод и вывод	Нет
<i>ncb_callname</i>	Необязательно	Нет
<i>neb_name</i>	Необязательно	Нет
<i>neb_rto</i>	Необязательно	Нет
<i>ncb_sto</i>	Необязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>neb_lana_num</i>	Ввод	Да
<i>ncbcmdcplt</i>	Вывод	Нет
<i>neb_event</i>	Ввод	Нет

## Команда *NCBRECVALY*

Команда получает данные от любого сеанса, соответствующего указанному имени, может быть использована и для получения данных, предназначенные для любого локального имени. Для этого нужно присвоить полю *nebjnum* значение 0xFF. В ином случае присвойте *nebjnum* номер сети, возвращенный с при добавлении имени в локальную таблицу имен. Затем любые данные, отложенные для данного имени, будут выбраны этой командой.

Если ожидают обработки несколько команд приема, действует тот же порядок приоритетов, что и при использовании команды *NCBRECVALY*.

Когда сеанс закрыт командой закрытия локального соединения, командой закрытия соединения на удаленной стороне или командой отмены соединения, любые ожидающие команды *NRCRECVALY* для указанного имени будут завершены с ошибкой *NRCSCLOSED* (0x0A). В поле *ncbjsn* из структуры *NCB* хранится номер локального соединения, которое было закрыто. Если отложенных для указанного имени команд *NCBRECVALY* нет, для закрытия того и какого-либо иного соединения вступает в силу отложенная команда: *NCBRECVALY* {*nebjnum* — 0xFF}. Она будет выполнена с ошибкой *NRCSCLOSED* а в поле *ncbjsn* — записан номер соответствующего сеанса.

Поле	Ввод или вывод	Нужно ли задать
<i>neb_command</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>neb_Isn</i>	Вывод	Нет

продолжение

Поле	Ввод или вывод	Нужно ли задать
<i>neb_num</i>	Ввод и вывод	Да
<i>ncbbuffer</i>	Ввод	Да
<i>ncbjlength</i>	Ввод и вывод	Да
<i>neb_callname</i>	Не обязательно	Нет
<i>neb_name</i>	Не обязательно	Нет
<i>neb_rto</i>	Не обязательно	Нет
<i>neb_sto</i> <i>M</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>neb_lana_num</i>	Ввод	Да
<i>neb_cmd_cplt</i>	Вывод	Нет
<i>ncb_event</i>	Ввод	Нет

## Команда **NCBRESET**

Команда сбрасывает указанный номер LANA и затрагивает определенные ресурсы среды.

*III* Если поле *ncbjsn* не равно 0, все ресурсы, связанные с *ncbjanaajium*, освобождаются.

*III* Если поле *ncbjsn* равно 0, все ресурсы, связанные с *ncbjanaajium*, освобождаются и выделяются новые. Байт *ncb\_callname[0]* задает максимальное количество сеансов, байт *ncb\_callname[2]* — максимальное количество имен, а байт *ncb\_callname[3]* требует, чтобы приложение использовало имя компьютера (у которого номер имени 1).

Поле	Ввод или вывод	Нужно ли задать
<i>nebjzommand</i>	Ввод	Да
<i>ncb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>nebjnum</i>	Ввод	Да
<i>nebjouffer</i>	Не обязательно	Нет
<i>ncbjlength</i>	Не обязательно	Нет
<i>ncb_callname</i>	Не обязательно	Нет
<i>nebjname</i>	Не обязательно	Нет
<i>ncbjrto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i>	Не обязательно	Нет
<i>ncbjanaajium</i>	Ввод	Да
<i>ncbcmdcjcpit</i>	Вывод	Нет
<i>neb_event</i>	Не обязательно	Нет

## Команда **NCBSEND**

Команда отправляет данные указанному участнику сеанса. Максимальный размер передаваемых данных — 65 536 байт (64 кб). Если удаленная сторона выдает команду завершения связи, все отложенные отправки возвращают ошибку, связанную с закрытием сеанса, — *NRCJSCLOSED* (0x0A). Отложенные команды отправки обрабатываются в порядке поступления.

Поле	Ввод или вывод	Нужно ли задать
<i>ncbcommand</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>neb_num</i>	Не обязательно	Нет
<i>nebbuffer</i>	Ввод	Да
<i>nebjlength</i>	Ввод	Да
<i>ncbcallname</i>	Не обязательно	Нет
<i>neb_name</i>	Не обязательно	Нет
с <i>neb_rto</i>	Не обязательно	Нет
<i>nebsto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>neb_lana_num</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb_event</i>	Ввод	Нет

## Команда **NCBSENDNA**

Команда отправляет данные указанному сеансу и не ожидает подтверждения от партнера. В остальном ведет себя так же, как *NCBSEND*.

Поле	Ввод или вывод	Нужно ли задать
<i>ncb_command</i>	Ввод	Нет
<i>ncb_retcode</i>	Вывод	Нет
<i>ncbjsn</i>	Ввод	Да
<i>nebjium</i>	Не обязательно	Нет
<i>nejbbuffer</i>	Ввод	Да
<i>nebjlength</i>	Ввод	Да
<i>ncb_callname</i>	Не обязательно	Нет
<i>nebjame</i>	Не обязательно	Нет
<i>ncbjrto</i>	Не обязательно	Нет
<i>ncb_sto</i>	Не обязательно	Нет
<i>ncb_post</i>	Ввод	Нет
<i>nebjanaajnum</i>	Ввод	Да
<i>ncb_cmd_cplt</i>	Вывод	Нет
<i>neb_event</i>	Ввод	Нет



# Команда **NCBSSTAT**

Команда получает сведения о состоянии сеанса. При вызове этой команды *ncb\_buffer* ссылается на область памяти, которая будет заполнена структурой *SESSION\_HEADER*, далее следуют одна или более структур *SESSION\_BUFFER*. Если первый байт *ncbjname* — звездочка (\*), команда выясняет состояние для всех сеансов, связанных со всеми именами в таблице локальных имен. Если предоставленный буфер слишком мал, выдается ошибка *NRCJNCOMP* (0x06). Если длина буфера меньше 4 байт — ошибка *NRCJBUFLN* (0x01).

Поле	Ввод или вывод	Нужно ли задать
<i>ncbjzommand</i>	Ввод	Да
<i>neb_retcode</i>	Вывод	Нет
<i>ncbjn</i>	Не обязательно	Нет
<i>neb_num</i>	Вывод	Нет
<i>neb_buffer</i>	Ввод	Да
<i>nebjlength</i>	Ввод	Да
<i>neb_callname</i>	Не обязательно	Нет
<i>ncbjname</i>	Ввод	Да
<i>neb_rto</i>	Не обязательно	Нет
<i>nebjsto</i>	Не обязательно	Нет
<i>nebjdost</i>	Ввод	Нет
<i>nebjana num</i>	Ввод	Да
<i>ncb_cmdjcplt</i>	Вывод	Нет
<i>neb_event</i>	Ввод	Нет

# Команда **NCBUNLINK**

Команда отменяет привязку к адаптеру. Предусмотрена для совместимости с ранними версиями NetBIOS и на платформах Win32 не оказывает эффекта.

# П Р И Л О Ж Е Н И Е

## Вспомогательные функции IP

*i*

В этом приложении мы опишем некоторые новые API-функции для получения информации и настройки параметров протокола IP. Эти функции позволяют программно использовать возможности следующих стандартных IP-утилит:

- **Ipconfig.exe** (или Winipcfg.exe в Microsoft Windows 95) — выводит информацию о конфигурации IP и позволяет освобождать и обновлять IP-адреса, выделенные DHCP;
- **Netstat.exe** — выводит таблицу TCP-соединений, таблицу прослушиваемых портов UDP и статистику протокола IP;
- **Route.exe** — выводит и редактирует таблицы маршрутизации;

**III Arp.exe** — выводит и редактирует используемые *протоколом разрешения адресов* (address resolution protocol, ARP) таблицы трансляции IP-адресов в физические.

Функции, описанные в этом приложении, доступны в основном в Windows 98 и Windows 2000, некоторые — и в Windows NT 4 с Service Pack 4 (или более поздним), но ни одна не доступна в Windows 95. При обсуждении каждой функции будет указываться соответствующая платформа. Прототипы для всех функций, описанных в приложении, определены в файле Iphlpapi.h. При компоновке приложения необходимо подключать библиотеку Iphlpapi.lib.

## Возможности утилиты Ipconfig

Утилита Ipconfig предоставляет два блока информации: о конфигурации IP и о параметрах IP для каждого сетевого адаптера, установленного на данном компьютере. Чтобы получить информацию о конфигурации IP, воспользуйтесь функцией *GetNetworkParams*:

```
DWORD GetNetworkParams(  
    PFIXED_INFO pFixedInfo,  
    PULONG pOutBufLen  
);
```

Параметр *pFixedInfo* — указатель на буфер, куда будет помещена структура *FIXED\_INFO* с информацией о конфигурации IP. Параметр *pOutBufLen* — указатель на переменную, задающую размер буфера, который передается через первый параметр. Если этого размера окажется не достаточно, функция

*GetNetworkParams* вернет значение *ERROR\_BUFFER\_OVERFLOW* и определит значение параметра *pOutBuJLen*, равным необходимому размеру буфера.

Структура *FIXED JNFO* имеет следующий вид:

S

```
typedef struct
{
    char            HostName[MAX_HOSTNAME_LEN + 4] ;
    char            DomainName[MAX_DOMAIN_NAME_LEN + 4] ;
    PIP_ADDR_STRING CurrentDnsServer;
    IP_ADDR_STRING  DnsServerList;
    UINT            NodeType;
    char            ScopeId[MAX_SCOPE_ID_LEN + 4] ;
    UINT            EnableRouting;
    UINT            EnableProxy;
    UINT            EnableDns;
} FIXEO_INFO, *PFIXED_INFO;
```

Она содержит следующие поля.

- *HostName* — имя данного компьютера, определенное системой DNS.
- III *DomainName* — домен DNS, к которому относится компьютер. #
- III *CurrentDnsServer* — IP-адрес текущего DNS-сервера.
- *DnsServerList* — связанный список, содержащий используемые данным компьютером DNS-серверы.
- II *NodeType* — способ, которым система распознает имена NetBIOS в сети с IP. Вот список его возможных значений:
  - III *BROADCASTNODETYPE* — разрешение имен NetBIOS типа б-узел; система использует IP-широковещание для регистрации и разрешения имен NetBIOS;
  - И *PEER\_TO\_PEER\_NODETYPE* — разрешение имен NetBIOS типа р-узел; система использует соединение «точка-точка» с сервером имен NetBIOS (например, WINS) для регистрации и разрешения имен компьютеров в IP-адреса;
  - К *MIXEDNODETYPE* — разрешение имен NetBIOS типа m-узел (mixed node, узел смешанного типа), при котором система использует оба описанных выше способа: сначала метод б-узла, если он не срабатывает, то метод р-узла;
  - *HYBRIDNODETYPE* — разрешение имен NetBIOS типа h-узел (hybrid node, гибридный узел); система также использует оба способа, но сначала — метод р-узла, а если он не срабатывает, то метод б-узла.
  - *Scopeld* — задает строку, добавляемую к имени NetBIOS для логического объединения двух и более компьютеров. Это необходимо для соединения NetBIOS по протоколу TCP/IP.
- Я *EnableRouting* — указывает, будет ли данная система осуществлять маршрутизацию IP-пакетов между сетями, к которым подключена.
- III *EnableProxy* — указывает, будет ли система работать в сети как прокси-агент системы WINS. Этот агент отвечает на широковещательные запросы

Я сы по именам, распознанными WINS, и позволяет сетям из компьютеров b-узлов соединяться с серверами в других подсетях, зарегистрированных в WINS.

- **EnableDns** — указывает, будет ли NetBIOS обращаться к DNS с именами, которые не разрешаются с помощью WINS, рассылки или файла *LMHOSTS*

Поле *DnsServerList* структуры *FIXED\_INFO* — это структура *IP\_ADDR\_STRING*, представляющая начало связанного списка IP-адресов:

```
typedef struct _IP_ADDR_STRING
```

```

    struct _IP_ADDR_STRING« Next;
*   IP_ADDRESS_STRING      IpAddress;                                i,  Q^
E   IP_MASK_STRING         IpMask;
    DWORD                  Context;      4 * '
} IP_ADDR_STRING, *PIP_ADDR_STRING;                                fry,      яy,

```

Поле *Next* содержит IP-адрес следующего в этом списке DNS-сервера. Если значение поля — *NULL*, то это означает конец списка. Поле *IpAddress* — строка символов, в которой IP-адрес представлен в десятично-точечной нотации. Поле *IpMask* — строка символов, в которой содержится маска подсети, соответствующая IP-адресу в поле *IpAddress*. Последнее поле — *Context*, связывает данный IP-адрес с уникальным для данной системы кодом.

Утилита `Ipconfig.exe` также получает информацию о конфигурации IP для конкретного сетевого оборудования. Это может быть адаптер Ethernet или даже адаптер удаленного доступа RAS. Информацию об адаптере дает функция *GetAdaptersInfo*:

```

DWORD GetAdaptersInfo (           ЦЩ,           %
    PIP_ADAPTER_INFO pAdapterInfo,
    PULONG pOutBufLen             "           "
);

```

Параметр *pAdapterInfo* — указатель на заранее размещенный в приложении буфер, в который будет помещена структура *ADAPTER\_INFO* с информацией о конфигурации адаптера. Параметр *pOutBufLen* должен содержать указатель на переменную, задающую размер этого буфера. Если размер не достаточен, функция *GetAdaptersInfo* вернет значение *ERRORBUFFER\_OVERFLOW* и передаст в параметре *pOutBufLen* требуемый размер буфера.

Структура *IP ADAPTER INFO* — это, фактически, список структур, содержащих информацию о настройках IP для каждого доступного на данном компьютере сетевого адаптера:

```
typedef struct IP_ADAPTER_INFO
```

```
struct _IP_ADAPTER_INFO* Next;
DWORD        ComboIndex;
char          AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
char          Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
UINT         AddressLength;
BYTE          Address[MAX_ADAPTER_ADDRESS_LENGTH];
```

```

DWORD      Index;
UINT       Type;
UINT       DhcpEnabled;
PIP_ADDR_STRING CurrentIpAddress;
IP_ADDR_STRING IpAddressList;
IP_ADDR_STRING GatewayList;
IP_ADDR_STRING DhcpServer;
BOOL       HaveWins;
IP_ADDR_STRING PrimaryWinsServer;
IP_ADDR_STRING SecondaryWinsServer;
time_t     LeaseObtained;
time_t     LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

Структура *IP\_ADAPTER\_INFO* содержит следующие поля.

- *Next* — следующий адаптер в буфере. Значение *NULL* означает конец списка.

*III CombolIndex* — не используется и задается равным 0.

- *AdapterName* — название адаптера.
- *Description* — краткое описание адаптера.

*III AddressLength* — размер физического адреса адаптера в байтах.

*Ж Address* — физический адрес адаптера.

- *Index* — уникальный код сетевого интерфейса, присвоенный данному адаптеру.

- *Type* — число, соответствующее типу адаптера:

- *MIBIFTYPEETHERNET* — адаптер Ethernet;
- *MIBIF TYPEFDDI* - адаптер FDDI;

*III MIB IFTYPELOOPBACK* — адаптер-петля;

- *MIBIF TYPE6THER* — иной тип адаптера;
- *MIBIF TYPE\_PPP* - адаптер PPP;
- *MIBIF TYPEJSIP* — адаптер SLIP;

*И MIBIFYPEJTOKENRING* — адаптер Token Ring.

*III DhcpEnabled* — показывает, задействована ли служба DHCP для этого адаптера.

*III CurrentIpAddress* — не используется и равен *NULL*.

*III IpAddressList* — список назначенных данному адаптеру IP-адресов.

- *GatewayList* — список IP-адресов шлюза по умолчанию.
- *DhcpServer* — список из одного пункта, содержащий IP-адрес используемого DHCP-сервера.
- *HaveWins* — показывает, использует ли этот адаптер WINS-сервер.
- *PrimaryWinsServer* — список из одного пункта, содержащий IP-адрес используемого основного WINS-сервера.

- Я ***SecondaryWititServer*** — список из одного пункта, содержащий IP-адрес используемого дополнительного WINS-сервера.
- В ***LeaseObtained*** — время получения аренды IP-адреса, выделенного DHCP-сервером.
- III ***LeaseExpires*** — дата окончания срока аренды IP-адреса.

## Освобождение и обновление IP-адресов

Утилита Ipconfig.exe может освобождать и обновлять IP-адреса с помощью параметров командной строки: /release и /renew. Для программного освобождения IP-адреса служит функция *IPReleaseAddress*:

```
DWORD IpReleaseAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
```

Для программного обновления IP-адреса применяется функция *IPRenewAddress*:

```
DWORD IpRenewAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
```

Единственный параметр этих функций — *AdapterInfo*, структура *IP\_ADAPTER\_INDEX\_MAP*, идентифицирующая адаптер, для которого надо освободить или обновить адрес

```
typedef struct _IP_ADAPTER_INDEX_MAP
```

```
    ULONG Index;
    WCHAR Name[MAX_ADAPTER_NAME];
}IP_ADAPTER_INDEX_MAP, *PIP_ADAPTER_INDEX_MAP;
```

Структура *IP\_ADAPTER\_INDEX\_MAP* содержит следующие поля.

- Ж ***Index*** — внутренний код сетевого интерфейса, присвоенный данному адаптеру;
- К ***Name*** — название адаптера.

Структура *IP\_ADAPTER\_INDEX\_MAP* для конкретного адаптера может быть получена с помощью функции *GetInterfaceInfo*:

```
DWORD GetInterfaceInfo (
    IN PIP_INTERFACE_INFO plfTable,
    OUT PULONG dwOutBufLen
```

Параметр *plfTable* — это указатель на буфер *IP\_INTERFACE\_INFO*, выделенный в приложении для информации об интерфейсе. Параметр *dwOutBufLen* — указатель на переменную, задающую размер этого буфера. Если размер недостаточен, функция *GetInterfaceInfo* вернет значение *ERRORINSUFFICIENTBUFFER* и передаст в параметре *dwOutBufLen* требуемый размер буфера.

Структура *IP\_INTERFACE\_INFO* определена так:

```
typedef struct _IP_INTERFACE_INFO
{
    LONG                      NumAdapters;
    IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO, *PIP_INTERFACE_INFO;
```

Она содержит следующие поля:

*III NumAdapters* — количество адаптеров в *поле Adapter*,

*III Adapter* — массив структур *IP\_ADAPTER\_INDEX\_MAP*.

Получив структуру *IP\_ADAPTER\_INDEX\_MAP* для конкретного адаптера, далее можно освободить или обновить выделенный DHCP-сервером IP-адрес с помощью функций *IPReleaseAddress* и *IPRenewAddress*.

## Изменение IP-адреса

Утилита *Ipsconfig.exe* не позволяет изменить IP-адрес для сетевого адаптера (кроме случая, когда используется служба DHCP). Однако имеются две вспомогательные IP-функции — *AddIpAddress* и *DeleteIpAddress*, позволяющие добавлять и удалять IP-адреса для адаптера. Применение этих функций требует знания внутренних кодов адаптеров и контекстных кодов IP-адресов. В Windows каждый сетевой адаптер имеет уникальный числовой код (ID), а каждый IP-адрес — уникальный контекстный код. Эти коды можно выяснить, вызвав функцию *GetAdaptersInfo*. Функция *AddIpAddress* определена так:

```
DWORD AddIpAddress (
    IPAddr Address,
    IPMask IpMask,
    DWORD IfIndex,
    PULONG NTEContext,
    PULONG NTEInstance
```

Параметр *Address* — это IP-адрес, добавляемый в виде длинного целого без знака. *IpMask* — маска подсети того же типа. *IfIndex* — порядковый номер адаптера, для которого добавляется адрес. Параметр *NTEContext* получает контекстный код, соответствующий добавляемому адресу, а параметр *NTEInstance* — соответствующий этому адресу код экземпляра.

Для программного удаления IP-адреса служит функция *DeleteIpAddress*:

```
DWORD DeleteIpAddress (
    ULONG NTEContext
);
```

Параметр *NTEContext* — это контекстный код, соответствующий удаляемому IP-адресу (его можно получить, вызвав функцию *GetAdaptersInfo*).

## Возможности утилиты *Netstat*

Эта утилита выводит информацию о таблице TCP-соединений, таблицу прослушиваемых портов UDP, а также статистику протокола IP для данного ком-

пьютера. Функции для получения этой информации работают не только в Windows 98 и Windows 2000, но также в Windows NT 4 с Service Pack 4 и последующих версиях.

## Получение таблицы TCP-соединений

Эта информация выводится на экран при запуске утилиты Netstat.exe с параметрами `-p tcp -a`. Чтобы получить эти данные программно, воспользуйтесь функцией *GetTcpTable*.

```
DWORD GetTcpTable(
    PMIBTCP_TABLE pTcpTable,
    PDWORD pdwSize,
    BOOL bOrder
```

Параметр *pTcpTable* — указатель на буфер *MIBTCP\_TABLE*, в который будет помещена информация о TCP-соединениях. Параметр *pdwSize* — указатель на переменную, задающую размер этого буфера. Если размера буфера недостаточно, функция передаст в этом параметре требуемый размер. Параметр *bOrder* указывает, нужно ли сортировать информацию.

Возвращаемая структура *MIBTCP\_TABLE* имеет вид:

```
typedef struct MIB_TCP_TABLE          ! B <

    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCP_TABLE, *PMIB_TCP_TABLE;
```

Она содержит следующие поля:

**Ж** *dwNumEntries* — количество строк в поле *table*;

**III** *table* — указатель на массив структур типа *MIBTCPROW*, содержащих информацию о TCP-соединениях.

Структура *MIBTCPROW* содержит информацию о парах IP-адресов, образующих TCP-соединение:

```
typedef struct _MIB_TCPROW
<
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;
```

Эта структура содержит следующие поля.

**III** *dwState* — состояние данного TCP-соединения. Может принимать значения:

- *MIBTCP\_STATE\_CLOSED* — CLOSED;
- III** *MIBTCP\_STATE\_CLOSING* - CLOSING;



- *MIB TCP STATE CLOSE WAIT* — CLOSE WAIT; *u*
  - III MIBTCP STATE DELETE TCB* — DELETE; *TM"*
  - III MIB TCP STATE ESTAB* — ESTABLISHED;
  - III MIB TCP STATE FIN WAIT 1* — FIN WAIT1;*f*
  - III MIB TCP STATEFINWAIT2* - FIN WAIT2; *^ ;<*
  - Ж MIB TCP STATE LAST ACK* — LAST ACK; *!*
  - III MIB TCP STATE LISTEN* — LISTENING;
  - *MIB TCP STATE SYNRCVD* — SYN RCVD;
  - *MIB TCP STATE SYNSENT*- SYN SENT;
  - *MIBTCPSTATE TIME WAIT* - TIME WAIT;
- III dwLocalAddr* — локальный IP-адрес данного соединения.
- II dwLocalPort* — локальный порт данного соединения.
- III. dwRemoteAddr* — внешний IP-адрес данного соединения.
- *dwRemotePort* — внешний порт данного соединения.

## Получение таблицы прослушиваемых портов UDP

Эта информация выводится на экран при запуске утилиты Netstat.exe с па>-  
раметрами -р udp -а. Чтобы получить эти данные, программно, воспользуй^  
тесь функцией *GetUdpTable*-.

```
DWORD GetUdpTable(
    PMIBJJDP_TABLE pUdpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

Параметр *pUdpTable* — указатель на буфер *MIBJJDP\_TABLE*, в который бу-  
дет помещена информация о прослушиваемых портах UDP. Параметр *pdw*-  
*Size* — указатель на переменную, задающую размер этого буфера. Если раз-  
мера буфера не достаточно, функция передаст в этом параметре требуемый  
размер. Параметр *bOrder* указывает, будет ли информация отсортирована.

Возвращаемая структура *MIBJJDP\_TABLE* имеет вид-

```
typedef struct _MIB_UDPTABLE
{
    DWORD dwNumEntries;
    MIBJJDP_ROW table[ANY_SIZE];
} MIB_UDPTABLE, * PMIB_UDPTABLE;
```

Она содержит следующие поля:

- III dwNumEntries* — количество строк в поле *table*;
- III table* — указатель на массив структур типа *MIBJJDP\_ROW*, содержащих  
информацию о прослушиваемых портах UDP.

Структура *MIBJJDP\_ROW* содержит IP-адрес, через который UDP ожидает  
приема дейтаграмм-

```
typedef struct _MIB_UDPROW
{
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, * PMIB_UDPROW;    i n i ^ j r    *ц*
```

Она содержит следующие поля:

- *dwLocalAddr* — локальный IP-адрес.
- *dwLocalPort* — локальный IP-порт.

## Получение статистики о протоколе IP

Статистическую информацию выводит утилита Netstat.exe с параметром -s. Ее можно получить также программно с помощью функций *GetIpStatistics*, *GetIcmpStatistics*, *GetTcpStatistics* и *GetUdpStatistics*. Функция *GetIpStatistics* получает статистику IP для данного компьютера:

S« ..      • «|

```
DWORD GetIpStatistics(
    PMIB_IPSTATS pStats
);
```

Параметр *pStats* — это указатель на структуру *MIB\_IPSTATS*, куда помещается статистика IP:

```
typedef struct _MIB_IPSTATS
{
    DWORD dwForwarding;
    DWORD dwDefaultTTL;
    DWORD dwInReceives;
    DWORD dwInHdrErrors;
    DWORD dwInAddrErrors;
    DWORD dwForwDatagrams;
    DWORD dwInUnknownProtos;
    DWORD dwInDiscards;
    DWORD dwInDelivers;
    DWORD dwOutRequests;
    DWORD dwRoutingDiscards;
    DWORD dwOutDiscards;
    DWORD dwOutNoRoutes;
    DWORD dwReasmTimeout;
    DWORD dwReasmReqds;
    DWORD dwReasmOks;
    DWORD dwReasmFails;
    DWORD dwFragOks;
    DWORD dwFragFails;
    DWORD dwFragCreates;
    DWORD dwNumIcf;
    DWORD dwNumAddr;
    DWORD dwNumRoutes;
} HIB_IPSTATS, *PMIB_IPSTATS;
```

— v

Структура *MIBIPSTATS* содержит следующие поля:

- III dwForwarding* — указывает, включено или нет на данном компьютере IP-перенаправление;
- III dwDefaultTTL* — начальное время TTL для отправляемых с данного компьютера дейтаграмм;
- III dwInReceives* — количество полученных дейтаграмм;
- III dwInHdrErrors* — количество дейтаграмм, полученных с ошибками в заголовках;
- *dwIttAddrErrors* — количество дейтаграмм, полученных с ошибками в адресах;
- *dwForwDatagrams* — количество перенаправленных дейтаграмм;
- *dwIriUnknownProtos* — количество дейтаграмм, полученных с неизвестными протоколами;
- *dwInDiscards* — количество отброшенных дейтаграмм;
- III dwInDelivers* — количество доставленных дейтаграмм;
- III dwOutRequests* — количество дейтаграмм с запросами на вывод;
- *dwRoutingDiscards* — количество отброшенных маршрутов;
- *dwOutDiscards* — количество отброшенных выходных дейтаграмм;
- *dwOutNoRoutes* — количество выходных дейтаграмм без маршрута;
- Н dwReasmtimeout* — максимальное время для приема фрагментированных дейтаграмм;
- III dwReasmReqds* — количество дейтаграмм, потребовавших сборки;
- *dwReasmOks* — количество успешно собранных дейтаграмм;
- *dwFragFails* — количество сбоях при фрагментации дейтаграмм;
- *dwFragCreates* — количество фрагментированных дейтаграмм;
- *dwNumIf* — количество доступных на данном компьютере IP-интерфейсов;
- *divNumAddr* — количество IP-адресов, назначенных данному компьютеру;
- III dwNumRoutes* — количество маршрутов в таблице маршрутизации.

Вторая функция — *GetIcmpStatistics*, служит для получения статистики протокола Internet Control Message Protocol (ICMP):

```
DWORD GetIcmpStatistics(
    PMIB_ICMP pStats
);
```

Параметр *pStats* — указатель на структуру *MIBICMP*, куда помещается статистика ICMP:

```
typedef struct _MIB_ICMP
{
    MIBICMPINFO stats;
} MIB_ICMP, *PMIB_ICMP;
```

Как видно, *MIBICMP* содержит структуру *MIBICMPINFO*-.

```
typedef struct _MIBICMPINFO
{
    MIBICMPSTATS icmpInStats;
    MIBICMPSTATS icmpOutStats;
} MIBICMPINFO;
```

Структура *MIBICMPINFO*, в свою очередь, содержит две структуры типа *MIBICMPSTATS*, *icmpInStats* — для статистики по входящей информации ICMP и *icmpOutStats* — для исходящей:

```
typedef struct .MIBICMPSTATS
{
    <                                     ;.-•
    DWORD dwMsgs;
    DWORD dwErrors;
    DWORD dwDestUnreachs;           ''
    DWORD dwTimeExcds;
    DWORD dwParmProbs;              «
    DWORD dwSrcQuenchs;
i  DWORD dwRedirects;              ... .
    DWORD dwEchos;
    DWORD dwEchoReps;
1  DWORD dwTimestamps;
i  DWORD dwTimestampReps;
    DWORD dwAddrMasks;
    DWORD dwAddrMaskReps;
} MIBICMPSTATS;
```

Структура *MIBICMPSTATS* содержит следующие поля:

- *dwMsgs* — количество посланных или полученных сообщений;
- III *dwErrors* — количество произошедших ошибок;
- *dwDestUnreachs* — количество сообщений о том, что адрес не доступен] i;
- III *dwTimeExcds* — количество превышений сроков;
- *dwParmProbs* — количество сообщений, содержащих неверные параметры IP;
- III *dwSrcQuenchs* — количество просьб снизить скорость;
- *dwRedirects* — количество переадресаций;
- III *dwEchos* — количество эхо-сообщений ICMP;
- *dwEchoReps* — количество ответов на эхо-сообщения;
- *dwTimestamps* — количество штампов времени;
- *dwTimestampReps* — количество ответов на штампы времени;
- III *dwAddrMasks* — количество масок адресов;
- *dwAddrMaskReps* — количество ответов на маски адресов.

Третья функция — *GetTcpStatistics*, служит для получения статистики протокола TCP:

```

DWORD GetTcpStatistics(          . JJ^J          \1,ОШЩ« ЯН>1
    PMIB_TCPSTATS pStats

```

Параметр *pStats* — указатель на структуру *MIBTCPSTATS*, куда помещается статистика:

```

typedef struct _MIB_TCPSTATS
{
    DWORD dwRtoAlgorithm;
    DWORD dwRtoMin;
    DWORD dwRtoMax;
    DWORD dwMaxConn;
    DWORD dwActiveOpens;
    DWORD dwPassiveOpens;
    DWORD dwAttemptFails;
    DWORD dwEstabResets;
    DWORD dwCurrEstab;
    DWORD dwInSegs;
    DWORD dwOutSegs;
    DWORD dwRetransSegs;
    DWORD dwInErrs;
    DWORD dwOutRsts;
    DWORD dwNumConns;
} MIB_TCPSTATS, *PMIB_TCPSTATS;

```

Структура *MIBTCPSTATS* содержит следующие поля:

- *dwRtoAlgorithm* — используемый алгоритм ретрансляции; допустимые значения: *MIBTCPRT0\_CONSTANT*, *MIB\_TCP\_RTO\_RSRE*, *MIB\_TCP\_RTO\_VANJ* и *MIB\_TCP\_RTO\_OTHER*;
- III *dwRtoMin* — минимальный лимит времени для ретрансляции в миллисекундах;
- III *dwRtoMax* — максимальный лимит времени для ретрансляции в миллисекундах;
- III *dwMaxConn* — максимально допустимое число подключений;
- III *dwActiveOpens* — количество подключений к серверу, инициированных данным компьютером;
- III *dwPassiveOpens* — количество подключений клиентов к данному компьютеру; \*
- *dwAttemptFails* — количество сбоев при подключении; • III
- III *dwEstabResets* — количество сброшенных подключений; \ III
- III *dwCurrEstab* — количество текущих подключений;
- *dwInSegs* — количество полученных сегментов;
- *dwOutSegs* — количество отправленных (в том числе повторно) сегментов;
- III *dwRetransSegs* — количество повторно отправленных сегментов;
- *dwInErrs* — количество сбоев при приеме;

III *dwOutRsts* — количество сегментов, переданных с флагом сброса;

III *dwNumConns* — общее количество подключений.

Последняя функция — *GetUdpStatistics*, служит для получения статистики протокола UDP:

```
DWORD GetUdpStatistics(
    FIB_UDPSTATS pStats
);
```

Параметр *pStats* — это указатель на структуру *MIBJJDPSTATS*, куда помещается статистика:

```
typedef struct MIBJJDPSTATS
{
    DWORD dwInDatagrams;
    DWORD dwNoPorts;
    DWORD dwInErrors;
    DWORD dwOutDatagrams;
    DWORD dwNumAddrs;
} MIB_UDPSTATS, *FIB_UDPSTATS;
```

Структура *MIBJJDPSTATS* содержит следующие поля:

III *dwInDatagrams* — количество полученных дейтаграмм;

III *dwNoPorts* — количество дейтаграмм, отброшенных из-за отсутствия портов;

III *dwInErrors* — количество ошибок при получении дейтаграмм (исключая *dwNoPorts*);

III *dwOutDatagrams* — количество переданных дейтаграмм;

- *dwNumAddrs* — общее количество записей в таблице прослушиваемых портов UDP.

## Возможности утилиты Route

Утилита *Route.exe* позволяет получать и редактировать таблицу маршрутов. Эта таблица определяет, какой из интерфейсов IP будет использован для обработки запроса или дейтаграммы. Библиотека вспомогательных IP-функций предлагает несколько функций для работы с таблицей маршрутов. Все они доступны в Windows 98, Windows 2000 и Windows NT 4 с Service Pack 4 (и выше).

Основное предназначение утилиты *Route.exe* — получение таблицы маршрутизации. Маршрут состоит из адреса подключаемого компьютера, маски подсети, шлюза, локального IP-интерфейса и метрики. С помощью утилиты можно также добавлять и удалять маршруты. Для добавления маршрута надо задать все перечисленные параметры, для удаления — только адрес подключенного компьютера.

## Получение таблицы маршрутов

Программно получить таблицу маршрутов можно с помощью функции *GetIpForwardTable*–.

```
DWORD GetIpForwardTable (                                1"
    PMIB_IPFORWARDTABLE pIpForwardTable,                ш
    PULONG pdwSize,
    BOOL border                                           (
);
```

Ее первый параметр — *pIpForwardTable*, это указатель на буфер в приложении, куда будет помещена таблица. Второй параметр — *pdwSize*, размер этого буфера. Если при вызове функции этот параметр равен *NULL* или размера буфера не достаточно, то в параметре *pdwSize* возвращается необходимый размер. Последний параметр — *bOrder*, указывает, будет ли информация отсортирована. По умолчанию информация сортируется так:

1. Адрес подключаемого компьютера.
2. Протокол маршрута.
3. Политика маршрутизации нескольких путей.

тв")  
i " "

4. Адрес следующего узла.

m

Информация о маршрутизации возвращается в виде структуры *MIB\_IPFORWARDTABLE*:

```
typedef struct _MIB_IPFORWARDTABLE                        ш
{
    i
    DWORD dwNumEntries;
    MIB_IPFORWARDROW table[ANY.SIZE];                    »
} MIB_IPFORWARDTABLE, *PMIB_IPFORWARDTABLE;              ц
```

Эта структура содержит массив структур *MIB\_IPFORWARDROW*. Поле *dwNumEntries* определяет их количество в этом массиве. Структура *MIB\_IPFORWARDROW* имеет вид:

```
typedef struct _MIB_IPFORWARDROW
{
    DWORD dwForwardDest;
    DWORD dwForwardMask;
    DWORD dwForwardPolicy;
    DWORD dwForwardNextHop;
    DWORD dwForwardIflIndex;
    DWORD dwForwardType;
    DWORD dwForwardProto;
    DWORD dwForwardAge;
    DWORD dwForwardNextHopAS;
    DWORD dwForwardMetric;
    DWORD dwForwardMetric2;
    DWORD dwForwardMetric3;
    DWORD dwForwardMetric4;
    DWORD dwForwardMetric5;
} MIB_IPFORWARDROW, *PMIB_IPFORWARDROW;
```

Структура *MIBJPFORWARDROW* содержит следующие поля:

*dwForwardDest* — IP-адрес конечного узла.

*dwForwardMask* — маска подсети конечного узла.

*dwForwardPolicy* — задает набор условий для выбора маршрута (см. RFC 1354), обычно представляемых в форме IP Type of Service (TOS).

*dwForwardNextHop* — IP-адрес следующего узла в маршруте.

*dwForwardIfIndex* — номер интерфейса для данного маршрута.

*divForwardType* — тип маршрута согласно RFC 1354; возможные значения:

*III MIB\_IPROUTE\_TYPEINDIRECT* — следующий узел не является конечным пунктом (дальний маршрут);

*III MIBIPROUTE TYPE DIRECT* — следующий узел является конечным пунктом (локальный маршрут);

« *MIBIPROUTETYPE INVALID* — неверный маршрут;

*III MIB IPROUTETYPEOTHER* — другой.

*dwForwardProto* — протокол маршрута; возможные значения (для протоколов IPX определены в файле Routprot.h, для IP — в файле Iprtrmib.h).

*II MIB IPPROTO OTHER* — неизвестный протокол;

*III MIBIPPROTOLOCAL* — маршрут, сгенерированный стеком;

*III MIBIPPROTONETMGMT* — маршрут, добавленный утилитой Route.exe или SNMP;

*K MIB IPPROTO ICMP* — маршрут от перенаправления ICMP;

*III MIB IPPROTO EGP* — протокол Exterior Gateway Protocol;

*IS MIBIPPROTOGGP* — протокол Gateway Gateway Protocol;

*И MIB IPPROTO HELLO* — протокол HELLO;

*III MIBIPPROTORIP* — протокол Routing Information Protocol;

• *MIBIPPROTOISJS* — соединение IP Intermediate System и Intermediate System Protocol;

*III MIBJPPROTOESIS* — соединение IP End System и Intermediate System Protocol;

*III MIBIPPROTO CISCO* - протокол IP Cisco;

*III MIBIPPROTO BBN* — протокол BBN;

*И MIB IPPROTOOSPF* — протокол Open Shortest Path First;

*В MIBIPPROTOBGP* — протокол Border Gateway Protocol;

*III MIB IPPROTONT AUTOSTATIC* — динамический маршрут, добавленный протоколом маршрутизации;

*III MIBIPPROTO NT STATIC* — маршруты, добавленные с помощью пользовательского интерфейса или утилитой Routemon.exe;

*III MIB IPPROTO STATIC NON DOD* — тоже, что и *PROTO\_IP\_NT\_STATIC*, кроме маршрутов, не вызывающих запрос Dial on Demand (DOD);

*И IPXPROTOCOLRIP* — протокол Routing Information Protocol for IPX;



Н *IPXPROTOCOLSAP* — протокол Service Advertisement Protocol;

• *IPXPROTOCOLNLSP* — протокол Netware Link Services Protocol. I

• *dwForwardAge* — время жизни маршрута в секундах. • •

III *dwForwardNextHopAS* — автономный системный номер следующего узла.

III *dwForwardMetric1* — зависящий от вида протокола параметр-метрика (подробнее — в RFC 1354); это значение отображается при запуске утилиты Route.exe с параметром print.

**ПРИМЕЧАНИЕ** Если этот параметр (а также следующие четыре) не используется, то его значение будет равно *MIBIPROUTE\_METRICJINUSED* (?).

• *dwForwardMetric2*, *dwForwardMetric3*, *dwForwardMetric4* и *dwForwardMetric5* — параметры, зависящие от вида протокола (подробнее — в RFC 1354).

Ч

## Добавление маршрута

Следующая функция утилиты Route.exe — добавление маршрута. Для этого необходимо задать IP-адрес подключаемого узла, маску подсети, шлюз, локальный интерфейс IP и метрику, проверив правильность параметров локального IP-интерфейса. Ссылаться на локальный IP-интерфейс надо по его внутреннему коду, который можно определить с помощью следующей функции *GetIpAddrTable*:

```
DWORD GetIpAddrTable (
    PMIB_IPADDRTABLE pIpAddrTable,
    PULONG pdwSize,
    BOOL border
);
```

Ее первый параметр — *pIpAddrTable*, указатель на буфер в приложении, куда будет помещена структура *MIBIPADDRTABLE*. Второй параметр — *pdwSize*, задает размер этого буфера. Если при вызове функции этот параметр задан равным *NULL* или размера буфера не достаточно, то в параметре *pdwSize* возвращается необходимый размер. Последний параметр — *bOrder*, указывает, будут ли локальные IP-интерфейсы отсортированы по возрастанию IP-адресов. Структура *MIBIPADDRTABLE* имеет вид:

```
typedef struct _MIB_IPADDRTABLE
{
    DWORD dwNumEntries
    MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE, *PMIB_IPADDRTABLE;
```

Эта структура содержит массив структур *MIBIPADDRROW*. Поле *dwNumEntries* определяет их количество в массиве. Код структуры *MIBIPADDRROW* следующий:

```
typedef struct _MIB_IPADDRROW
```

```

    DWORD    dwAddr;
    DWORD    dwIndex;
    DWORD    dwMask;
    DWORD    dwBCastAddr;
    DWORD    dwReasmSize;
    unsigned short unused1;
    unsigned short unused2;
MIB_IPADDRROW, *PMIB_IPADDRROW;
```

Структура *MIB\_IPADDRROW* содержит следующие поля:

- ***dwAddr*** — IP-адрес для данного интерфейса;
- ***dwIndex*** — код данного интерфейса;

III ***dwMask*** — маска подсети для данного IP-адреса;

III ***dwBCastAddr*** — адреса рассылки;

III ***dwReasmSize*** — максимальный размер сборки для принимаемых дейтаграмм;

III ***unused1* и *unused2*** — не используются.

С помощью этой функции можно определить, правильны ли параметры локального IP-интерфейса. Чтобы добавить маршрут в таблицу, воспользуйтесь функцией *SetIpForwardEntry*:

```

DWORD SetIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

Ее единственный параметр — *pRoute*, указатель на структуру *MIB\_IPFORWARDROW*. Для добавления маршрута должны быть заданы значения полей *dwForwardIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop* и *dwForwardPolicy*.

## Удаление маршрута

Удаление маршрута — простейшая операция *Route*. Для этого надо задать только адрес подключавшегося компьютера. Соответствующая этому адресу структура *MIB\_IPFORWARDROW* возвращается функцией *GetIpForwardTable* и передается функции *DeleteIpForwardEntry*, удаляющей маршрут:

```

DWORD DeleteIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

Можно вручную задать необходимые поля параметра *pRoute* — *dwForwardIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop* и *dwForwardPolicy*.

## Утилита ARP

Эта утилита используется для просмотра и редактирования кэша ARP. Пример из Platform SDK, эмулирующий ее работу с помощью вспомогательных

функций IP, называется *Iprgr.exe*. Протокол ARP отвечает за разрешение IP-адреса в физический MAC-адрес. Для увеличения производительности компьютер хранит эту информацию в кэше, и к ней можно получить доступ с помощью утилиты *Agr.exe*. Утилита позволяет вывести таблицу ARP (с параметром *-a*), удалить запись (с параметром *-d*) или добавить запись (с параметром *-s*).

Рассмотрим соответствующие функции. Они доступны в Windows 98, Windows 2000 и Windows NT 4 с Service Pack 4 (и выше).

Получить таблицу ARP позволяет функция *GetIpNetTable*:

```
DWORD GetIpNetTable (
    PMIB_IPNETTABLE pIpNetTable,
    PULONG          pdwSize,
    BOOL            border
);
```

Ее первый параметр — *pIpNetTable*, указатель на буфер в приложении, куда будет помещена структура *MIBIPNETTABLE*. Вторым параметром — *pdwSize*, размер этого буфера. Если при вызове функции этот параметр равен *NULL* или размера буфера недостаточно, то в параметре *pdwSize* возвращается необходимый размер буфера (при этом сама функция возвращает значение *ERRORINSUFFICIENTBUFFER*). Последний параметр — *bOrder*, указывает, будут ли записи отсортированы по возрастанию IP-адресов. Структура *MIB\_IPNETTABLE* имеет следующий вид:

```
typedef struct _MIB_IPNETTABLE
{
    DWORD          dwNumEntries;
    MIB_IPNETROW  table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

Эта структура содержит массив структур *MLBIPNETROW*. Поле *Entries* определяет их количество в этом массиве. Каждая структура *NETROW* содержит одну запись ARP:

```
typedef struct _MIB_IPNETROW {
    DWORD dwIndex;
    DWORD dwPhysAddrLen;
    BYTE  bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
} MIB_IPNETROW, *PMIB_IPNETROW;
```

Структура *MLB LPNETROW* содержит следующие поля.

**К** *dwIndex* — код адаптера.

**П** *dwPhysAddrLen* — длина физического адреса в байтах.

- *bPhysAddr* — массив байт, содержащий физический (MAC) адрес адаптера.
- *dwAddr* — IP-адрес адаптера.
- *dwType* — тип записи ARP; возможные значения:

- Ж MIBJPNETJYPE STATIC* — статическая запись;
- Ш MIBJPNETJTYPEJDYNAMIC* — динамическая запись;
- *MIBJPNETJYPEJNVAUD* — недействительная запись;
- Ш MIB\_IPNET\_TYPE\_OTHER* — иной тип.

## Добавление записи ARP

Для добавления записи ARP предназначена функция *SetIpNetEntry*:

```
DWORD SetIpNetEntry (
    PMIB_IPNETROW pArpEntry
```

Ее единственный аргумент — рассмотренная структура *MIBJPNETROW*, в которой должны быть заданы необходимые поля. В поле *dwIndex* указывается код локального IP-адреса сети, к которой применяется запись ARP. Этот код для каждого IP-адреса можно определить с помощью функции *GetIpAddrTable*. Следующее поле — *dwPhysAddrLen*, обычно равно 6. (Большинство физических адресов, в частности, MAC-адреса ETHERNET занимают 6 байт) Массив байт *bPhysAddr* содержит физический адрес. Большинство MAC-адресов представляются в виде 12 символов и должны быть переведены в соответствующий двоичный вид. Например MAC-адрес 00-A0-C9-A7-86-E8 будет представлен как

```
00000000 10100000 11001001 10100111 10000110 11101000
```

Метод кодирования аналогичен применяемому для кодирования адресов IPX и ATM (см. главу 6). Поле *dwAddr* должно содержать IP-адрес узла, для которого задавался MAC-адрес. В последнем поле — *dwType*, указывается тип записи. После заполнения необходимых полей структуры вызывается функция *SetIpNetEntry*. Если запись успешно добавлена, возвращается значение *NOERROR*.

## Удаление записи ARP

Для удаления записи требуется только задать в структуре *MIBJPNETROW* код интерфейса *dwIndex* и IP-адрес удаляемой записи *dwAddr*. Затем вызывается функция *DeletIpNetEntry*.

```
DWORD DeletIpNetEntry (
    PMIB_IPNETROW pArpEntry
```

Напомним, что код для локального интерфейса IP можно получить с помощью функции *GetIpAddrTable*. Если удаление записи успешно, функция *DeletIpNetEntry* возвращает значение *NOJIRRROR*.

# П Р И Л О Ж Е Н И Е

## Коды ошибок Winsock

В приложении перечислены коды ошибок Winsock, отсортированные по номерам. В список не включены специфичные и недокументированные ошибки Winsock. Ошибки Winsock, непосредственно соответствующие ошибкам Win32, перечислены в конце приложения.

**Прерванный вызов функции.** Блокирующий вызов прерывается вызовом *WSACancelBlockingCall*.

*10009-WSAEBADF*

**Неверный дескриптор файла.** В ОС Microsoft Windows CE такую ошибку может вернуть функция *socket*, она означает, что общий последовательный порт занят.

*10013-WSAEACCES*

**Доступ запрещен.** Была сделана попытка обращения к запрещенному сокету. Обычно эта ошибка возникает при попытке использовать широко-вещательный адрес в функциях *sendto* или *WSASendTo*, когда широковещание не разрешено параметрами *setsockopt* и *SOJ3ROADCAST*.

*10014-WSAEFAULT*

**Недопустимый адрес.** Функции Winsock передан недопустимый указатель адреса. Ошибка также возникает, когда указан слишком маленький буфер.

*10022-WSAEINVAL*

**Недопустимый параметр.** Указано недопустимое значение параметра, например, контрольного кода в функции *WSAIoc!*. Эта же ошибка может возникнуть из-за текущего состояния сокета, например, при вызове функций *accept* или *WSAAccept* на сокете, не находящемся в состоянии прослушивания.

*W024-WSAEMFILE*

**Открыто слишком много файлов.** Открыто слишком много сокетов. Обычно системы доступа Microsoft ограничены только количеством доступных ресурсов ОС.

*10035-WSAEWOULDBLOCK*

**Ресурс временно не доступен.** Ошибка наиболее часто возвращается неблокирующими сокетами, на которых запрошенную операцию (например, вызов функции *connect*) нельзя выполнить немедленно.

*10036-WSAEINPROGRESS*

**Операция выполняется.** Блокирующая операция находится в процессе выполнения. Ошибка появляется редко, только при разработке 16-битных приложений Winsock.

*10057-WSAEALREADY*

**Действие уже выполняется.** На неблокирующем сокете происходит попытка выполнить операцию, которая уже вызвана и выполняется например, при повторном вызове на неблокирующем сокете функции *connect* или *WSAConnect* во время установления соединения. Ошибка также возникает, когда поставщик службы находится в процессе выполнения функции обратного вызова (для функций, поддерживающих обратный вызов)

*10058-WSAENOTSOCK*

**Операция на недопустимом сокете.** Ошибка может быть возвращена любой функцией Winsock параметром которой является дескриптор типа *SOCKET*. Означает, что задан неверный дескриптор сокета

*WO59-WSAEDESTADDRREQ*

**Требуется адрес назначения.** Ошибка появляется, если пропущен адрес например, при вызове функции *sendto* с адресом назначения *INADDR\_ANY*

*10040-WSAEMSGSIZE*

**Слишком длинное сообщение.** Сообщение отправляется на дейтаграммный сокет размером больше внутреннего буфера. Ошибка также возникает, если сообщение больше, чем позволяют ограничения сети. Еще одна причина появления данной ошибки — буфер слишком мал, чтобы вместить полученную дейтаграмму

*10041-WSAEPROTOTYPE*

**Неверный тип протокола для сокета.** При вызове *socket* или *WSASocket* указан протокол, не поддерживающий семантику данного типа сокета. Например, попытка создать IP-сокет типа *SOCKSTREAM* с протоколом *IPPROTOUDP*

*WO42-WSAENOPROTOOPT*

**Неверный параметр протокола.** Неизвестный, неподдерживаемый или неверный параметр сокета. Ошибка также возникает, если уровень указан при вызове *getsockopt* или *setsockopt*

*10045-WSAEPROTONOSUPPORT*

**Не поддерживаемый протокол.** Запрошенный протокол не установлен в системе. Например, попытка создать TCP- или UDP-сокет, при том, что в системе не установлен протокол TCP/IP

*10044-WSAESOCKTNOSUPPORT*

**Не поддерживаемый тип сокета.** Данное семейство адресов не поддерживает указанный тип сокета. Например, запрос сокета типа *SOCK\_RAI* по протоколу, не поддерживающему простые сокеты

*10045-WSAEOPNOTSUPP*

**Не поддерживаемая операция.** Объект обращения не поддерживает запрошенную операцию. Обычно возникает при попытке вызвать функции Winsock на сокете, не поддерживающем данную операцию. Например, *accept* или *WSAAccept* на дейтаграммном сокете

*10046-WSAEPFNOSUPPORT*

**Не поддерживаемое семейство протоколов.** Запрошенное семейство протоколов не существует или не установлено в системе. В большинстве случаев ошибка означает то же, что и более частая — *WSAEAFNOSUPPORT*

10047- *WSAEAFNOSUPPORT*

**Семейство адресов не поддерживает запрошенную операцию.** Попытка выполнить операцию, не поддерживаемую данным типом сокета: например, при вызове *sendto* или *WSASendTo* на сокете типа *SOCKJSTREAM*. Ошибка также возникает при вызове *socket* или *WSASocket* с указанием неверной комбинации семейства адресов, типа сокета и протокола.

*W048-WSAEADDRINUSE*

**Адрес уже используется.** При обычных обстоятельствах только один сокет может использовать каждый адрес сокета. Например, адрес IP-сокета состоит из локального IP-адреса и номера порта. Ошибка обычно связана с функциями *bind*, *connect* и *WSAConnect*. Для функции *setsockopt* может быть задан параметр *SO\_REUSEADDR*, разрешающий использование несколькими сокетами одного и того же локального IP-адреса и порта (см. главу 9)-

10049-*WSAEADDRNOTAVAIL*

**Невозможно назначить запрошенный адрес.** При вызове API указан недопустимый для данной функции адрес: например, в функции *bind* задан IP-адрес, не относящийся к локальному IP-интерфейсу. Также ошибка может возникнуть при указании порта 0 удаленной машины, с которой производится соединение, в функциях *connect*, *WSAConnect*, *sendto*, *WSASendTo*, и *WSAJoinLeaf*.

10050-*WSAENETDOWN*

**Сеть не работает.** Операция не может быть выполнена из-за неполадок в сети, стеке, сетевом интерфейсе или с локальной сети.

10051-*WSAENETUNREACH*

**Сеть недоступна.** Попытка произвести операцию в недоступной сети: локальный узел не знает, как достичь удаленный. Другими словами, не существует известного маршрута к месту назначения.

10052-*WSAENETRESET*

**Сеть разорвала соединение при сбросе.** Соединение было разорвано, но из-за невозможности доставки сообщений о его продолжении. Ошибка также возникает при попытке задать параметр *SO\_KEEPLIVE* в функции *setsockopt*, когда соединение уже разорвано.

10053-*WSAECONNABORTED*

**Преждевременный разрыв соединения из-за ошибки ПО.** Произошла ошибка протокола или истек таймаут.

10054-*WSAECONNRESET*

**Соединение разорвано партнером по связи.** Установленное соединения было принудительно закрыто удаленным узлом. Ошибка возникает, если удаленный процесс не работоспособен (например, при ошибке обращения к памяти или сбое аппаратуры) или если на сокете было произведено принудительное закрытие. Сокет можно настроить для резкого закрытия с помощью параметра *SO\_JINGER* и функции *setsockopt* (см. главу 9)-

10055-*WSAENOBUFS*

**Свободное местовбуфере закончилось.** Запрошенная операция не может быть произведена, так как системе не хватает свободного места в буфере.

*10056-WSAEISCONN*

**Сокетом уже установлено соединение.** Попытка установить соединение с сокетом, который уже используется. Ошибка может произойти как на дейтаграммных, так и на потоковых сокетах. При использовании дейтаграммного сокета ошибка возникает при вызове *sendto* или *WSASendTo*, еон для установления дейтаграммного соединения вызывались функции *connect* или *WSAConnect*.

*10057-WSAENOTCONN*

**Нет соединения с сокетом.** Производится запрос на отправку или получение данных на сокете, соединение с которым в данный момент не установлено.

*10058-WSAESHUTDOWN*

**Отправление невозможно после отключения сокета.** Сокет уже частично закрыт вызовом функции *shutdown*, но к нему делается запрос на отправление или получение данных. Ошибка возникает только на отключенных направлениях потока данных-, например, при попытке отправить данные после вызова *shutdown*.

*10060-WSAETIMEDOUT*

**Время ожидания операции истекло.** Сделан запрос на соединение но удаленный компьютер не отвечает должным образом (или вообще не отвечает) по истечении определенного промежутка времени. Ошибка обычно возникает, когда заданы параметры сокета *SO\_SNDTIMEO* и *SORCVTIMEC* или при вызове функций *connect* и *WSAConnect* (см. главу 9)-

*10061-WSAECONNREFUSED*

**В соединении отказано.** Компьютер-адресат отказывает в установлении соединения, обычно из-за того, что на удаленной машине нет приложения, обслуживающего соединение с указанным адресом.

*10064-WSAEHOSTDOWN*

**Узел не работает.** Попытка выполнить операцию не удалась, так как узел назначения выключен. Тем не менее, приложение, скорее всего, получит ошибку *WSAETIMEDOUT*, обычную при попытке установить соединение.

*10065-WSAEHOSTUNREACH*

**Не известен путь к узлу.** Был произведен запрос на выполнение операции на недоступном узле. Эта ошибка похожа на *WSAENETUNREACH*.

*10067-WSAEPROCM*

**Слишком много процессов.** Некоторые поставщики служб WinsocJ задают предел количества процессов, которые могут обращаться к ним одновременно.

*10091-WSASYSNOTREADY*

**Подсистема сети не доступна.** Ошибка возвращается при обращении к *WSAStartup*, когда поставщик службы не может отработать из-за того, что базовая система, предоставляющая услуги, не доступна.

*10092-WSAVERNOTSUPPORTED*

**Некорректная версия Winsock.dll.** Запрошенная версия поставщика службы Winsock не поддерживается.



*10095-WSANOTINITIALISED*

**Winsock не инициализирован.** Вызов *WSAStartup* не удался.

*10101-WSAEDISCON*

**Идет корректное завершение работы.** Ошибка возвращается функциями *WSARecv* и *WSARecvFrom*, указывая, что удаленный партнер находится в процессе корректного завершения работы. Возникает при использовании протоколов, ориентированных на передачу сообщений, таких, как ATM.

*10102-WSAENOMORE*

**Более записей не найдено.** Ошибка возвращается функцией *WSALookupServiceNext* и означает, что больше нет дополнительных записей. Она аналогична ошибке *WSA\_E\_NO\_MORE*. Приложения должны проверять возврат обеих этих ошибок.

*10W3-WSAECANCELLED*

**Операция отменена.** Ошибку возвращает *WSALookupServiceNext*, если во время отработки этой функции был вызов *WSALookupServiceEnd*. Аналогична *WSA\_E\_CANCELLED*. Приложения должны проверять возврат обеих ошибок.

*10104-WSAEINVALIDPROCTABLE*

**Неверная таблица вызовов процедуры.** Ошибка обычно возвращается поставщиком службы, если в таблице процедур содержатся недопустимые записи (см. главу 14).

*10105-WSAEINVAUDPROVIDER*

**Недопустимый поставщик службы.** Поставщик не может установить корректную версию Winsock, необходимую для правильного функционирования.

*101Q6-WSAEPROVIDERFAILEDINIT*

**Поставщик службы не инициализирован.** Поставщик не может загрузить требуемые библиотеки (DLL).

*10107-WSASYSCALLFAILURE*

**Сбой системного вызова.** Неблагоприятный исход системного вызова, который не должен давать сбой.

*10108-WSASERVICE\_NOT\_FOUND*

**Не найдена такая служба.** Запрошенная служба не найдена в данном пространстве имен. Ошибка обычно возникает при работе функций регистрации и разрешения имен при запросе служб (см. главу 10).

*10109-WSATYPE\_NOT\_FOUND*

**Не найден тип класса.** Ошибка также относится к функциям регистрации и разрешения имен. Когда регистрируется экземпляр службы, то он должен ссылаться на класс службы, заданный ранее функцией *WSAInsta/lServiceClass*.

*10110-WSA\_E\_NO\_MORE*

**Записей более не найдено.** Ошибка возвращается функцией *WSALookupServiceNext* и аналогична *WSAENOMORE*. Приложения должны проверять возврат обеих этих ошибок.

^ 10111 -WSA\_E\_CANCELLED

**4 Операция отменена.** Ошибку возвращает *WSALookupServiceNext*, если во время отработки этой функции был вызов *WSALookupServiceEnd*. Аналогична *WSAECANCELLED*. Приложения должны проверять возврат обеих этих ошибок.

10112-WSAEREFUSED

**Запрос отклонен.** Запрос к базе данных не удался.

11001-WSAHOSTNOTFOUND

**Узел не найден.** Ошибку возвращают *gethostbyname* и *gethostbyaddr*, указывая, что полномочный узел не найден^

1 \002-WSATRY\_AGAIN

**Неполномочный узел не найден.** Не был найден неполномочный узел, либо произошла ошибка в работе сервера. Ошибка связана с *gethostbyname* и *gethostbyaddr*.

\1005-WSANO\_RECOVERY

**Произошла неустраняемая ошибка.** Ошибка связана с *gethostbyname* и *gethostbyaddr*. Следует попытаться выполнить операцию еще раз.

11004-WSANOJDATA

**Не найдено записей данных запрошенного типа.** Не найдено записей данных запрошенного типа, хотя заданное имя было верным. Ошибка связана с *gethostbyname* и *gethostbyaddr*.

11005-WSA\_QOS\_RECEIVERS

**Получено минимум одно сообщение резервирования.** Минимум один процесс в сети хочет получать трафик Quality of Service (QoS). Значение связано с реализацией QoS в IP и, по сути, ошибкой не является (см. главу 12.).

1 \006-WSA\_QOS\_SENDERS

**Получено минимум одно сообщение пути.** Минимум один процесс в сети хочет отправлять трафик QoS. Значение является сообщением о состоянии.

11007-WSA\_QOS\_NO\_SENDERS

**Нет отправителей QoS.** Более не существует процессов, которые хотели бы отправлять данные QoS (см. главу 12).

1 W08-WSA\_QOS\_NO\_RECEIVERS

**Нет получателей QoS.** Более не существует процессов, которые хотели бы получать данные QoS (см. главу 12).

1 \009-WSA\_QOS\_REQUEST\_CONFIRMED

**Запрос на резервирование подтвержден.** Утвердительный ответ на запрос о резервировании пропускной способности, который могут выдавать приложения QoS (см. главу 12).

11010-WSA\_QOS\_ADMISSION\_FAILURE

**Ошибка из-за недостаточности ресурсов.** Не хватило ресурсов для удовлетворения запроса на пропускную способность QoS.

11011 -*WSA\_QOS\_POLICYJAILURE*

**Неверные реквизиты.** Пользователь не владеет необходимыми полномочиями, или заданные реквизиты не позволяют выполнить запрос резервирования QoS.

11012-*WSA\_QOS\_BAD\_STYLE*

**Неизвестный или противоречивый стиль.** Приложения QoS могут задавать разные стили фильтрации для данного сеанса (см. главу 12).

11013-*WSA\_QOS\_BAD\_OBJECT*

**Неверная структура *FILTERSPEC* или специфичный для поставщика объект.** Ошибка в структуре *FILTERSPEC* или специфичных для поставщика буферах объекта QoS (см. главу 12).

1 \Q\A-*WSA\_QOS\_TRAFFIC\_CTRL\_ERROR*

**Проблемы с *FLOWSPEC*.** У компонента управления трафиком появилась проблема с параметрами *FLOWSPEC*, переданными в составе объекта QoS.

11015-*WSA\_QOS\_GENERIC\_ERROR*

**Общая ошибка QoS.** Универсальная ошибка, возвращаемая, когда не применимы другие ошибки QoS.

6-*WSA\_INVALID\_HANDLE*

**Указан неверный объект события.** Функции *WSAWaitForMultipleEvents* передан неверный описатель. Ошибка Win32, появляется при использовании функций Winsock, соответствующих функциям Win32.

8-*WSA\_NOT\_ENOUGH\_MEMORY*

**Не достаточно памяти.** Для выполнения операции не хватает свободной памяти, ошибка Win32.

87-*WSA\_INVALID\_PARAMETER*

**Один или несколько неверных параметров.** Функции передан неверный параметр, ошибка Win32. Также возникает при работе *WSAWaitForMultipleEvents*, когда задан неверный параметр счетчика событий.

258-*WSA\_WAIT\_TIMEOUT*

**Таймаут операции истек.** Перекрытая операция не завершилась в положенное время, ошибка Win32.

995-*WSA\_OPERATION\_ABORTED*

**Перекрытая операция прервана.** Перекрытая операция ввода-вывода отменена из-за закрытия сокета, ошибка Win32. Также возникает при выполнении команды *SIOFLUSH*.

996-*WSAJOINCOMPLETE*

**Объект события для перекрытого ввода-вывода не свободен.** При вызове *WSAGetOverlappedResults* перекрытая операция ввода-вывода не завершена, ошибка Win32.

997-*WSA\_IO\_PENDING*

**Перекрытая операция завершится позже.** При перекрытом вызове ввода-вывода операция отложена и завершится позже, ошибка Win32 (см. главу 8).

## \ От авторов

Мы хотим поблагодарить всех, кто внес свой вклад в создание этой книги.

Особой благодарности заслуживает Вей Хуа (Wei Hua), разработавший все примеры Visual Basic для прилагаемого компакт-диска. Вей также консультировал нас по поводу портов завершения ввода-вывода и поставщиков транспортных служб в протоколе интерфейса поставщика служб (SPI) Winsock 2.

Спасибо Берри Баттеркли (Barry Butterklee) за пояснения по вопросам, касающимся портов завершения ввода-вывода. Берри также проверил главу < QoS и много сделал, чтобы информация в ней была правильной и точной

Амол Дешпенд (Amol Deshpande) предоставил ценную информацию об описателях устанавливаемой файловой системы (IFS) многоуровневых поставщиков служб Winsock 2. Он также помог нам как эксперт по отладке программ при разработке примера пакета данных, описывающих состояние сети (link-state packet, LSP).

Фрэнк Ли (Frank Li) предоставил подлинный пример регистрации и разрешения имен (RNR), который послужил основой главы 10. Фрэнк также разработал некоторые примеры IP Helper, вошедшие в состав прилагаемого к книге компакт-диска.

Арвинд Мерчинг (Arvind Murching) и Аншуль Дхир (Anshul Dhir) проявили многие вопросы, касающиеся поставщика ATM для Winsock.

Алекс Чоу (Alex Choe), разбирающийся практически во всех современных сетевых протоколах, помог нам с описанием деталей функционирования протокола IPX/SPX, вошедшим в главу о семействе адресов Winsock.

Кен Эванс (Ken Evans) описал и объяснил многие структуры данных ДБ интерфейса прикладного программирования IP Helper, определение которых приведено в Приложении В.

Фрэнк Ким (Frank Kim) и Дэвид Моуэрс (David Mowers) помогли рассказать о системе безопасности Windows NT, которая используется перенаправителем Windows, почтовыми ящиками и именованными каналами.

Мэтт Ниблер (Matt Nibler) выверил технические сведения о перенаправителе Windows, которому посвящена глава 2.

Гэри Юкиш (Gary Yukish) оказал помощь в пояснении возможностей сервера удаленного доступа.

Сахин Кукрейя (Sachin Kukreja) ответил на многие вопросы о QoS.

И наконец, Мазахир «Маази» Пунавала (Mazahir "Maazaazy" Poonawala) предоставил пример RAS для компакт-диска.

Мы также благодарны группе по работе с документацией Microsoft Platform SDK за их великодушное и активное участие. Хотим выразить признательность Ребекке Маккей (Rebecca McKay) и Донни Камерон (Donnie Cameron) из Microsoft Press. Редактирование Ребекки сделало книгу доступной для читателя. Донни занималась выверкой точности технической информации прекрасно справилась с этим нелегким делом.

# Предметный указатель

access control entity *см* ACE  
 access control list *см* ACL  
 access token *см* маркер доступа  
 ACE (access control entity) 59  
   — SID 60  
   — запрещающая доступ 60  
   — разрешающая доступ 60  
 ACL (access control list) 59  
 ACS (Admission Control Service) 346, 348, 396  
 Active Directory 297  
 address resolution protocol *см* ARP  
 Admission Control Service *см* ACS  
 ADSP 201  
 AESA (NSAP-style ATM Endsystem Address) 150  
 APC (Asynchronous Procedure Call) 200, 445  
 API (application programming interface) 2  
 AppleTalk 111, 117, 287  
   — Wmsock 140, 202  
   — адресация 140  
   — имя 140-143  
   — параметр 148  
   — протокол 148  
   — сокет 255  
 application programming interface *см* API  
 ARP (address resolution protocol) 549, 565  
 Arp exe 549  
 ASCII 455  
 Asynchronous NetBEUI 501  
 Asynchronous Procedure Call *см* APC  
 Asynchronous Transfer Mode *см* ATM  
 ATM (Asynchronous Transfer Mode) 111, 117, 148  
   — ioctl-команда 283  
   — NNI 149  
   — QoS 396  
   — SAP 150  
   — UNI 149  
   — Winsock 204  
   — адрес 152  
   — корневой узел 315  
   — листовой учел 315  
   — многоадресная рассылка 331  
   — поставщик 153  
   — преобразование строки 152  
   — разрешение имени 155  
   — сокет 153  
   — схема адресации 150

## В

### BBN 563

Berkeley Socket *см* сокет Беркли  
 BHLI (Broadband Higher Layer Information) 150  
 BLII (Broadband Lower Layer Information) 150

BLOB 304  
 Border Gateway Protocol 563  
 Broadband Higher Layer Information *см* BHLI  
 Broadband Lower Layer Information *см* BLII  
 broadcasting *см* широковещание

## CHAP 515

Client for Microsoft Networks *см* клиент для сетей Microsoft  
 Client Services for NetWare 288  
 completion port *см* порт завершения  
 control plane *см* многоадресная рассылка плоскость управления

DACL (Discretionary Access Control List) 59, 83

data plane *см* многоадресная рассылка плоскость данных

datagram *см* дейтаграмма

Delegation *см* делегирование

device identifier *см* идентификатор устройства

Dial on Demand *см* DOD

dial-up networking *см* DUN

Discretionary Access Control List *см* DACL

DNS (Domain Name System) 126 286, 304

DOD (Dial on Demand) 563

Domain Name System *см* DNS

DUN (dial-up networking) 501 *см также* RAS клиент

E164 150

EAP 519

Ethernet 148

Exterior Gateway Protocol 563

Fault Tolerant Distributed Server *см* сервер отказоустойчивый распределенный

FQDN (Fully Qualified Domain Name) 126

FTP (File Transfer Protocol) 122

Gateway Gateway Protocol 563

Generic Packet Classifier *см* GPC

Generic Quality of Service *см* GQoS

globally unique identifier *см* GUID

GPC (Generic Packet Classifier) 347

GQoS (Generic Quality of Service) 343

GQoS API 346

GUID (globally unique identifier) 276, 289, 290, 373

# Н

host-byte-order *см* системный порядок следования байт

HTTP (Hypertext Transfer Protocol) 122

## I

I/O redirection *см* перенаправление ввода-вывода

IANA (Internet Assigned Numbers Authority) 123

IAS (Information Access Service) 128-129, 131, 260, 261

ICMP (Internet Control Message Protocol) 118, 398, 400, 558

ICS (Internet connection service) 519

Identification *см* идентификация

IGMP (Internet Group Management Protocol) 311-312, 342, 398, 412

Impersonation *см* олицетворение

Information Access Service *см* IAS

Infrared Data Association *см* IrDA

Infrared socket *см* IrSock

Infrared Sockets 117, 118

Integrated Services *см* IntServ

Intermediate System Protocol 563

Internet Assigned Numbers Authority *см* IANA

Internet connection service *см* ICS

Internet Control Message Protocol *см* ICMP

Internet Group Management Protocol

*см* IGMP

Internet Protocol *см* IP

Internetwork Packet Exchange *см* IPX

interprocess communication *см* IPC

IntServ (Integrated Services) 365

ioctl-команда

- *FIONBIO* 273

- *FIONREAD* 274

- *SIO\_ADDRESSLISTCHANGE* 281

- *SIO\_ADDRESSLISTQUERY* 280

- *SIO\_ASSOCIATE\_JVC* 284

- *SIO\_CHKJOS* 276

- *SIOJNABILE\_CIRCULAR\_QUEUEING* 274»

- *SIO\_FIND\_ROUTE* *lib*

- *SIOJLUSH* 215

- *SIO\_GET\_ATM\_ADDRESS* 284

- *SIO\_GET\_ATM\_CONNECTIONID* 284 ?

- *SIO\_GET\_BROADCAST\_ADDRESS* 275 "

- *SIOSETJXTENSIONJUNCTION\_*

*POINTER* 275

- *SIO\_GETINTERFACE\_LIST* 281

- *SIO\_GET\_NUMBER\_OF\_ATM\_DEVICES* 283

- *SIO\_GET\_QOS* 111

- *SIOJEEPALIVEJVALS* 278

- *SIO\_MULTICASTSCOPE* 111

- *SIO\_MULTIPOINTJOOPBACK* 111, 342

- *SIOJICVALL* 278

- *SIOJICVALL\_IGMPMAST* 279

- *SIOJCVALLJMAST* 279

- *SIO\_ROUTINGINTERFACE\_CHANGE* 279

- *SIO\_ROUTINGINTERFACEJQUERY* 279

- *SIO\_SET\_QOS* 277

- *SIOATMARK* 274

- *SOJSL\_GET\_CAPABILITYES* 282

- *SOJSL\_GETJLAGS* 282

- *SOSSLJ3ETJROTocols* 282

- *SO\_SSL\_PERFORM\_HANDSHAKE* 283

- *SOJSLJETJLAGS* 282

- *SO\_SSL\_JET\_PROTOCOLS* 283

- *SOJSLJETJALIDATE\_CERT\_HOOK* 283

IP (Internet Protocol) 110, 116, 121, 286-287, 501

- конфигурация 549

- листовой узел 313

- рассылка 314

- сокет 262

- статистика 557

IP Cisco 563

IP End System 563

IP Intermediate System 563

Ipapexe 566

IPC (interprocess communication) 64, 78

Ipconfigexe 549

Ipndncexe 423

IPv4 122

IPv6 122

IPX (Internetwork Packet Exchange) 133, 287, 501

- адаптер 269, 271

- дейтаграмма 269

- заголовок 269

- номер сети 270, 272

- маршрутизатор 133

- расширенная адресация пакетов 269

- тип пакета 268

- фильтрация типов пакетов 268

- широкощательный пакет 272

IPX/SPX 2, 4, 110, 116, 203, 268

IP-адрес 6, 122, 286

- изменение 554

- обновление 553

- освобождение 553

- разрешение имени 125

- специальный 123

IP-адресация 399

IrDA (Infrared Data Association) 128-120, 258

IrDA Logical Service Access Point Selector

*см* LSAP-SEL

IrSock (Infrared socket) 128-129, 258

## К

keepalive *см* TCP сохранение соединени!

keepalive packet *см* пакет сообщений об активности

L2TP (Layer 2 Tunneling Protocol) 519

LAN 121

LAN Adapter *см* LANA

LAN Manager *см* диспетчер ЛВС

LANA (LAN Adapter) 4, 542

- Windows 9x 52

- диапазон 4

- проверка состояния 50

- таблица имен 17

LANE 314

Laver 2 Tunneling Protocol *см* L2TP  
 Local Policy Module *см* LPM  
 loop *см* петля  
 LPM (Local Policy Module) 348  
 LSAP-SEL (IrDA Logical Service Access Point Selector) 128,262

**M**

MAC (media access control) 345  
 MAC-адрес 49  
 mailslot *см* почтовый ящик  
 Mailslot File System *см* MSFS  
 marshaling data *см*marshaling данных  
 master browser *см* координатор сети  
 Max Response Time *см* максимальное время ожидания  
 Maximum Segment Lifetime *см* MSL  
 maximum transmission unit *см* MTU  
 Mcastatm *с* 331  
 Mcastws2 *с* 326  
 media access control *см* MAC  
 Microsoft Networking Provider *см* MSNP  
 MSFS (Mailslot File System) 65  
 MSL (Maximum Segment Lifetime) 165  
 MSNP (Microsoft Networking Provider) 57, 78  
 MTU (maximum transmission unit) 267, 278  
 multicast address *см* групповой адрес  
 multicasting *см* многоадресная рассылка  
 MUP (Multiple UNC Provider) 54, 55

**N**

Nagle 105  
 Name Binding Protocol *см* NBP  
 name space provider *см* NSP  
 named pipe *см* именованный канал  
 Named Pipe File System *см* NPFS  
 NBP (Name Binding Protocol) 140, 255, 287  
 Nbtstat 6  
 NCB (network control block) 9  
 NCP (NetWare Core Protocol) 136  
 NDS (NetWare Directory Services) 287-288  
 NetBEUI (NetBIOS Extended User Interface) 2-3,4,501  
 NetBIOS 111, 117  
 — LANA 4  
 — LanEnum 16  
 — Netapi32hb 9  
 — ResetAU 17  
 — Win32 2  
 — Windows 95 52  
 — Windows 98 52  
 — Windows CE 52  
 — Windows NT 4.0 4  
 — WINS 6  
 — Winsock 137, 202  
 — адресация 137  
 — асинхронный вызов 11  
 — виртуальный канал 8  
 — дейтаграмма 34, 35  
 — заголовочный файл 9  
 — зарегистрированное имя 6

— имя 5, 137  
 — групповое 6, 7  
 — добавление 18  
 — удаление 18  
 — уникальное 6  
 — клиент 30  
 — поверх TCP/IP 112  
 — приложение 3  
 — протокол 3  
 — разрешение имени 7, 138  
 — сеанс 8  
 — синхронный вызов 11  
 — служба 8  
 — спецификатор 7  
 — таблица имен 5, 6  
 NetBIOS (Network Basic Input/Output System) 2  
 NetBIOS API 3, 9  
 NetBIOS Configuration 5  
 NetBIOS Extended User Interface *см* NetBEUI  
 NetBT 4  
 Netstatex 549,554  
 Netware 287  
 NetWare 133  
 NetWare Core Protocol *см* NCP  
 NetWare Directory Services *см* NDS  
 Netware Link Services Protocol 564  
 Network Basic Input/Output System ,  
*см* NetBIOS  
 network control block *см* NCB  
 network interface card *см* NIC  
 network node interface *см* NNI  
 network number *см* номер сети  
 network operating system *см* NOS  
 network provider *см* сетевой поставщик  
 Netwoik Service Access Point *см* NSAP  
 network-byte order *см* сетевой порядок следования байт  
 NIC (network interface card) 345  
 NNI (network node interface) 149  
 NNTP 265  
 node number *см* номер узла  
 NOS (network operating system) 54, 56  
 Novell Client v3.01 for Windows 95/98 *см* клиент для сетей Novell  
 NPFS (Named Pipe File System) 79  
 NSAP (Network Service Access Point) 150  
 NSAP-style ATM Endsystem Address *см* AESA  
 NSP (name space provider) 426  
 — удаление 453, 464  
 — установка 464  
 Null DACL 86  
 NWLink IPX/SPX/NetBIOS Compatible Transport Protocol 3  
 OOB (out-of-band) 170,251,274  
 Open Shortest Path First 563  
 Open Systems Interconnect *см* OSI  
 option header *см* расширенный заголовок  
 OSI (Open Systems Interconnect) 2, 119  
 out-of-band *см* OOB  
 overlapped I/O *см* перекрытый ввод-вывод

- Packet Scheduler 347
- Packet Shaper 347, 358
- PAP (Password Authentication Protocol) 201, 515
- partial message см фрагментарное сообщение *Ч*
- Password Authentication Protocol см PAP
- PDU (Protocol Data Unit) 262
- PE (Policy Element) 348
- per I/O-operation data см данные операции ввода-вывода
- per-handle data см сокет данные описателя
- permanent virtual circuit см PVC
- Ping 398, 401
- Plug-and-Play 4
- Point-to-Point Protocol см PPP
- Point-to-Point Tunneling Protocol см PPTP
- Policy Element см PE
- PPP (Point-to-Point Protocol) 501 *т*
- PPTP (Point-to-Point Tunneling Protocol) 519 *т*
- primary login см основной реквизит входа?
- Protocol Data Unit см PDU (*Ч*)
- PVC (permanent virtual circuit) 284
  
- QoS (Quality of Service) 108, 148, 163, 343
  - ACS 348
  - ATM 396
  - IP-приоритет 345
  - LPM 348
  - PE 348
  - RSVP 361
  - Traffic Control API 347
  - Winsock 2 348
  - Winsock API 396
  - диспетчер SBM 346
  - класс трафика 358
  - локальная система 346
  - место назначения 359
  - многоадресный UDP 395
  - модуль TC 347
  - одноадресный TCP 373
  - одноадресный UDP 394
  - одноадресное соединение 352
  - оповещение на канальном уровне 346
  - очередь формирователя 360
  - передача данных 344
  - поставщик службы GQoS 346
  - приоритет 357
  - сквозное соединение 346
  - стандарт 802.1p 345
  - шаблон 371
- QOS 213, 349
- Quality of Service см QoS
  
- RAS (Remote Access Service) 50, 342
  - IP-соединение 534
  - клиент 501
  - код ошибки 508
  - описатель соединения 531, 532
  - пакет обновления DUN 1.3 503
  - подключение 502, 508
  - приложение 502
  - реквизит безопасности пользователя 527
  - структура данных 503
  - телефонный справочник 514 523 529
- raw socket см сокет простой
- redirector см перенаправитель
- Remote Access Service см RAS
- request ID см идентификатор запроса
- Resource Reservation Protocol см RSVP
- RESV 361
- RIP (Routing Information Protocol) 136 563
- Rlogin 170
- Route.exe 549, 561
- Routing Information Protocol см RIP
- Routing Information Protocol for IPX 563
- Routing Table Maintenance Protocol см RTMP
- RSVP (Resource Reservation Protocol) 276, 344-346, 360
  - инициирование сеанса 365
  - информация о состоянии 360
  - многоадресная рассылка 368
  - объект 364
  - одноадресный TCP 367
  - одноадресный UDP 366
  - политика 365
  - резервирование 361
- RTMP (Routing Table Maintenance Protocol) 140
  
- SACL (system access control list) 59, 83
- SAP см Service Access Point или Service Advertising Protocol
- SBM (Subnet Bandwidth Manager) 345, 346
- scatter-gather I/O см комплексный ввод-вывод
- security descriptor см дескриптор безопасности
- Security Identifier см SID
- Sequenced Packet Exchange см SPX
- Sequencer см секвенсор
- Serial Line Internet Protocol см SLIP
- Server Message Block см SMB
- Server Message Block File Sharing Protocol 57
- Service Access Point (SAP) 150
- Service Advertisement Protocol 564
- Service Advertising Protocol (SAP) 136, 287
- service class см служба класс
- Shiva's Password Authentication Protocol см SPAP
- SID (Security Identifier) 59, 60, 83
- SLIP (Serial Line Internet Protocol) 501
- SMB (Server Message Block) 54, 57, 65
- Socket Information 491
- SPAP (Shiva's Password Authentication Protocol) 515
- SPX (Sequenced Packet Exchange) 133, 136



SPXII 134

SSL 282

State Information 491

Subnet Bandwidth Manager *см* SBMsystem access control list *см* SACL

TAPI (Telephony Application Programming Interface) 502

TC (Traffic Control module) 346

TCP (Transmission Control Protocol) 121

— QoS 373

— клиент 489

— приложение

—запуск 490

—пример 480-481

— сервер 487, 488

— сокет 268, 491

— сохранение соединения 278

— таблица соединений 555

TCP/IP 2, 4, 112, 118, 122

— Nagle 105

— Windows CE 200

— размер окна 173

Telephony Application Programming Interface *см* TAPI

Telnet 170

time-to-live *см* TTL

TOS (type of service) 345

Traceroute 398, 411

Traffic Control (TC) API 346

Traffic Control module *см* TC

Traffic Shaper 359

Transmission Control Protocol *см* TCP

TSP (transport service provider) 426, 427

— счетчик экземпляров 433

— установка 447

— базовый (base) 427

— многоуровневый (layered) 427

— функция поддержки 429

TTL (time-to-live) 165, 277, 312, 354, 411

type of service *см* TOS**И**

UDP (User Datagram Protocol) 122, 398, 401

— QoS 391, 395

— пакет 411

— инициализация заголовка 423

— приложение

—запуск 478

—окно 472

—пример 473

— простой сокет 415

— псевдозаголовок 415

— сообщение

—пересылка 476

—прием 477

— таблица прослушиваемых портов 556

— формат заголовка 414

UDP/IP 122

UNC (Universal Naming Convention) 54, 55, 79

UNI (user network interface) 149

UNICODE 455

Universal Naming Convention *см* UNCUser Datagram Protocol *см* UDPuser network interface *см* UNI

Uuidgen.exe 290

VBCE (Visual Basic Toolkit for Windows CE) 494

VC (Virtual Connection) 148

virtual device driver *см* VXD

Visual Basic 468

Visual Basic Toolkit for Windows CE *см* VBCE

VPN 519

VXD (virtual device driver) 53

**W**

WAN (wide area network) 121

Win32 2, 287

Windows Internet Naming Server *см* WINS

Windows Open System Architecture

*см* WOSA

Wimpcfg.exe 549

WINS (Windows Internet Naming Server) 6

Winsock 469

— AppleTalk 202

— ATM 204

— IP 122

— IrDA 202

— NetBIOS 202

— OOB 170

— QoS 348, 371

— TCP 480

— UDP-сообщение 476

— Windows CE 200

— библиотека 157

— блокирующий ввод-вывод 436

— информация о состоянии 478, 490

— клиент 160

— код ошибки 568

— конфигурация 426

— массив элементов 481

— метод 470

— многоадресная рассылка 316

— модель ввода-вывода

—select 209, 437

—WSAAsyncSelect 213, 439

—WSAEventSelect 217, 441

—перекрытого 223, 442

—портов завершения 234

— номер версии 158

— обработка ошибок 159

— обратный вызов поставщика 426

— ошибка 491

— параметр 340-341

— платформа 159

— процедура завершения 230

— разрешение имени 287

— регистрация имени 287

— регистрация службы 293

— режим работы сокета 205

— сервер 160

— событие 471

— состояние TCP 164

- элемент управления 494
- Winsock 1 112, 118, 158, 316, 317
- Winsock 2 52, 104, 158, 286
  - ioctl-команда 274
  - WOSA-архитектура 425
  - многоадресная рассылка 323
  - отладка 466
- Winsock 2 DLL 158
- Winsock 2 SDK 118
- Winsock 2 (SPI) 425, 426
- Winsock DLL 114
- Winsock Information 478, 490
- WOSA (Windows Open System Architecture) 425
- WPU 426
- WSC 426
- WSP 426

архитектура клиент-сервер 67, 78  
 асинхронный вызов процедур *см* APC  
 аудит 59

безопасность 61—62  
 блок
 

- данных 350
- данных протокола *см* PDU
- сетевую управления *см* NCB

 буква локального диска 55

## B

виртуальное соединение *см* VC  
 время жизни *см* TTL

глобальная сеть *см* WAN  
 глобально уникальный идентификатор *см* GUID  
 групповое имя домена 7  
 групповой адрес 311

## D

данные операции ввода-вывода 238  
 дейтаграмма 34
 

- IPX 269
- блокирующий прием 45
- отправка 34
- прием 35, 46, 192
- размер 50, 158

 делешрование 95  
 дескриптор безопасности 59
 

- DACL 59
- SACL 59
- SID 60

 диспетчер
 

- перекрытого ввода-вывода 442
- LBC 57

 домен 65  
 доменная система имен *см* DNS  
 драйвер виртуального устройства *см* VXD

запись управления доступом *см* ACE

## I

идентификатор
 

- безопасности *см* SID
- запроса 489
- локального диска 56
- устройства 128

 идентификация 95  
 избирательный список управления доступом *см* DACL  
 именованный канал 54 78
 

- дескриптор безопасности 84
- клиент 80, 95, 97
- код ошибки 80
- перекрытый ввод-вывод 90
- поток 88
- режим передачи данных 79
- режим работы 99
- сервер 80, 85
- фла! режима создания 81
- фла! режима чтения-записи 83
- формат имени 79
- экземпляр 81

 интерфейс прикладного программирования *см* API  
 инфракрасные сокет *см* InfraredSocket

## K

качество обслуживания *см* QoS  
 клиент
 

- для сетей Microsoft 55, 57 *яв*
- для сетей Novell 55
- удаленного доступа *см* DUN

 команда
 

- асинхронная 11
- блокирующего действия 24
- блокирующей передачи 24
- опроса состояния адаптера 48
- поиска имени 48, 51
- синхронная 11

 коммутатор сет и 345  
 комплексный ввод-вывод 169, 175  
 компонент сетевого доступа 55 *см также* селевой поставщик  
 контроллер домена 7  
 концентратор сети 345  
 координатор сети 7

## M

максимальная единица передачи *см* MTU  
 максимальное время жизни сегмента *см* MSL  
 максимальное время ожидания 413  
 максимальный блок передачи данных *см* MTU  
 маркер доступа 59, 61  
 маршалинг данных 463  
 маршрут
 

- добавление 564
- удаление 565

маршрутизация 109  
 межпроцессная связь *см* IPC  
 многоадресная рассылка 6, 108, 308  
 — ATM 314 331  
 — IGMP 312 412  
 — IP-интерфейс 341  
 — RSVP 368  
 — TTL 340  
 — Winsock 1 1 316 317  
 — Winsock 2 323  
 — в сетях IP 311, 313  
 — плоскость  
 ———корневая 308  
 ———маршрутизируемая 309, 310  
 ———{«маршрутизируемая 309, 310  
 ———равноправная 309  
 ———данных 308  
 —управления 308  
 — средствами Winsock 1 313  
 — таблица маршрутов 341  
 модель событий 11, 24  
 модуль  
 — локальной полигики *см* LPM  
 — управления трафиком *см* TC

## Н

негарантированный трафик 343  
 номер  
 — сетевого адаптера *см* LANA  
 — сети 133, 136  
 — узла 133  
 обратная совместимость 2  
 объект 357  
 олицетворение 94  
 операция чтения 68  
 описатель события 11  
 основной реквизит входа 61  
 открытые системы Windows *см* WOSA

## П

пакет сообщений об активности 250  
 перекрытый ввод-вывод 69, 90, 169, 205  
 перенаправитель 54, 57, 61, 65  
 — SMB 58  
 — клиент 58  
 — компонент 56  
 — сети Microsoft *см* MSNP  
 перенаправление ввода-вывода 54  
 петля 140  
 плата сетевого интерфейса *см* NIC  
 политика безопасности 347  
 полное имя домена *см* FQDN  
 порт  
 — локальный 473  
 — удаленный 473  
 — завершения 205  
 ———корректное закрытие 241  
 ———настройка 236  
 ———перекрытый ввод-вывод 238  
 ———рабочий поток 235, 240  
 ———создание 234  
 ———сокет 235

поставщик нескольких UNC *см* MUP  
 постоянный виртуальный канал связи \*  
*см* PVC  
 поток 88  
 — приоритет 357  
 — сквозной 344  
 почтовый ящик 54 64  
 — время ожидания 76  
 — дейтаграмма 65  
 — запись данных 71  
 — имя 65, 68  
 — клиент 67, 70, 71  
 — код ошибки 67  
 — описатель 67  
 — передача сообщения 66  
 — правило  
 ———<8 3> 73  
 —именования 70  
 — право доступа 68  
 — размер сообщения 66, 68  
 — реализация 67  
 — сервер 67, 69  
 — создание 68  
 — утечка памяти 76  
 — чтение данных 69  
 программный интерфейс компьютерной /  
 телефонии *см* TAPI )  
 пространство имен 287, 455  
 — DLL 462  
 — DNS 287, 304  
 — NDS 287 )  
 — NTDS 296  
 — SAP 296 \$  
 — Win32 287  
 — динамическое 287  
 — запрос 466  
 — иерархичное 300  
 — поставщик 289  
 — постоянное 287 j  
 — предопределенное 287  
 — служба 464  
 ———класс 289  
 ———регистрация 289  
 ———удаление 293  
 — сохранение данных 462 ||  
 — список 287  
 — статическое 287  
 — целочисленная константа 289  
 — целочисленное значение 287  
 протокол  
 — Win32 110  
 — Winsock 2 112  
 — атрибут адресации 287  
 — гарантированная доставка 106  
 — кадрирующий (framing) 501  
 — корректное завершение работы 107  
 — маршрутизация 109  
 — многоадресное вещание 108  
 — ориентированный на передачу  
 сообщений 104  
 — основанный на потоке (stream-  
 based) 104  
 — порядок доставки 106  
 — потоковый 173  
 — пространство имен 287

- фрагментарное сообщение 109
- широковещание 108
- разрешения адресов *см* ARP
- резервирования ресурсов *см* RSVP
- процедура обратного вызова 11
- псевдопоток 106

- разрешение имени 125, 286
- расширенный заголовок 411
- регистрация имени 286
- реквизит сеанса 61

- связь между процессами *см* IPC
- сеанс 8

- закрытый 18
- максимальное количество 50
- номер 18

- секвенсор 358

- сервер 160

- имя 164
- отказоустойчивый
- распределенный 286

- сетевая операционная система *см* NOS
- сетевой порядок следования байт 123, 124

- сетевой поставщик 55 *см также*
- компонент сетевого доступа

- системный адрес ATM в сгиле NSAP
- см* AESA

- системный порядок следования байт 124
- системный список управления доступом
- см* SACL

- служба

- DNS-запрос 304

- IFS-поставщик 434

- IP-адрес 297

- LSAP-SEL-номер 132

- SAP Agent 288

- базовый поставщик 274

- дейтаграммная 8, 9, 105

- запрос 299, 301

- класс 289, 291

- регистрация 457

- удаление 458, 465

- установка 465

- номер порта 122, 127, 290

- поставщик

- идентификатор каталога 435

- многоуровневый 449

- пространства имен 453

- удаление 452

- упорядочение 451

- базовый 448

- пространства имен *см* NSP

- транспорта *см* TSP

- потоковая 105

- регистрация 289, 293

- сеанс 8

- соединений с Интернетом *см* ICS

- создание 289

- удаление 293

- удаленной о доступа *см* RAS

- управления допуском *см* ACS
- событие 471

- сокет 116

- AppleTalk 148, 255

- ATM 153

- IP 125, 262

- IPX 134

- IPX/SPX 268

- IrDA 258

- IrSock 133

- NetBIOS 139

- SPX 270

- TCP 125, 165, 176, 250, 268

- UDP 125, 176, 415, 479

- Беркли 175

- данные описателя 235

- дейтаграммный 139, 274

- дескриптор 162

- закрытие 176

- значение QoS 355

- инфракрасного канала *см* IrSock

- модель 205

- неблокирующий 324

- описатель 433

- освобождение ресурсов 189

- отправка данных 187

- очередь входящих сообщений 277

- пересылка данных 168

- порт завершения 235

- потоковый 153, 267

- привязка 135, 160

- привязка к SAP 150, 154

- прием данных 168, 171, 185

- простой 118, 398

- размер буфера 242, 252

- режим 205

- блокирующий (blocking) 205, 206

- неблокирующий (nonblocking) 205, 206, 273

- прослушивания 160

- семейство адресов 117

- сервера 165

- соединение клиентов 161

- создание 125, 133-134, 139

- состояние 165

- тип 134

- сокетная пара 166

- список управления доступом *см* ACL

- срочные данные *см* OOB

- таблица

- имен 535

- маршрутизации 549

- маршрутов 562

- тип службы *см* TOS

- точка доступа

- к сетевым службам *см* NSAP

- доступа к службам *см* SAP

- трафик

- приоритет 344

уведомление

- *bd\_QOS* 369
- *RSVP* 369
- *WSA\_QOS\_NO\_RECEIVERS* 371
- *WSA\_QOS\_NO\_SENDERS* 371
- *WSA\_QOS\_RECENsRS* 371
- *WSA\_QOS\_REQUEST\_CONFIRMED* 371
- *WSA\_QOS\_SENDERS* 371

универсальное правило именования *см* UNC

управление доступом к среде *см* MAC  
устройство инфракрасной связи *см* IrDA

## Ф

файл

- UNC-имя 55
- доступ 54
- создание 62

формирователь

- пакетов *см* Packet Shaper
  - трафика *см* Traffic Shaper
- фрагментарное сообщение 109

центр безопасности 61

## Ш

широковещание 6, 108, 247, 308, 479  
широкополосная информация верхнего  
уровня *см* BHLI  
широкополосная информация нижнего  
уровня *см* BLII

элемент политики *см* PE

эмулирующая локальная сеть *см* LANE



## Энтони Джонс

Энтони Джонс (Anthony Jones) родился в Сан-Антонио, Техас. В 1996 г окончил с отличием Техасский университет, получив ученую степень бакалавра по информатике и вычислительной технике. Затем занимался научной работой по оптимизации компилятора Icon.

Некоторое Энтони работал в «Southwest Research Institute» — некоммерческой исследовательской компании в Сан-Антонио. Там он принял участие в нескольких проектах, включая разработку встроенных управляющих систем реального времени и ПО визуализации и моделирования для самых разных заказчиков — от BBC США до ТВ-программы «Weather Channel». В 1997 г Энтони переехал в Вашингтон, где был включен в группу NetAPI для работы в составе Microsoft Developer Support. Недавно он перешел в отдел Windows 2000 Core Networking, где в настоящее время занимает должность испытателя в группе Winsock.

В свободное время Энтони увлекается горным велоспортом, лыжами, туризмом, фотографией. Охотно смотрит программу «Futurama» и телесериал «The X-Files».



## Джим Оланд

Джим Оланд (Jim Ohlund) — инженер-разработчик ПО в группе тестирования Microsoft Proxy Server в Ричмонде, Вашингтон. Он успел поработать во многих областях компьютерной промышленности: был системным программистом, тестировал ПО.

В 1990 г Джим получил ученую степень бакалавра по информатике и вычислительной технике в Техасском университете. Джим начал заниматься компьютерами, еще будучи студентом колледжа: именно тогда он создал систему работы с персоналом для Министерства обороны США. Он расширил свои знания компьютерных сетей в 1994 г, разрабатывая программное обеспечение эмуляции терминала для платформ Windows. В 1996 г Джим вошел в команду Microsoft Developer Support Networking API, помогая разработчикам ПО применять на практике сетевые API, о которых рассказывается в этой книге.

Когда Джим не занят работой с компьютерами, он любит кататься на лыжах, на велосипеде и путешествовать по Тихоокеанскому побережью.

# ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

(прилагаемый к кнше компакт-диск)

**ЭТО ВАЖНО - ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО.** Настоящее лицензионное соглашение (далее «Соглашение») является юридическим документом, оно заключается между Вами (физическим или юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, то не имеете права устанавливать, копировать или как-то иначе использовать этот Программный Продукт.

## ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными до1 опорами по авторскому праву, а также дру1 ими законами и договорами по правам на интеллектуальную собственность

### 1. ОБЪЕМ ЛИЦЕНЗИИ. Настоящее Соглашение дает Вам право

а) **Программный продукт.** Вы можете установить и использовать одну копию Программного Продукта на одном компьютере. Основной пользователь компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использовать ее на портативном компьютере

б) **Хранение или использование в сети.** Вы можете также скопировать или установить экземпляр Программного Продукта на устройстве хранения, например на сетевом сервере, исключительно для установки или запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно на других компьютерах

с) **License Pak.** Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того, Вы получаете право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше

д) **Примеры кода.** Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее <Примеры >), если таковые входят в состав Программного Продукта

и) **Использование и модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (d)(in) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию

ii) **Распространяемые файлы.** При соблюдении пункта (d)(ii) Microsoft дает Вам право на свободное от отчислений копирование и распространение в виде объектно! о кода Примеров или их модифицированной версии, кроме тех частей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению

ш) **Требования к распространению файлов.** Вы можете распространять файлы, разрешенные к распространению при условии, что а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть, б) не используете название, эмблему или товарные знаки Microsoft для продвижения своего приложения, в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и заставки своего приложения, и) со1 ласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения, и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft

## 2. ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ

• **Ограничения на реконструкцию, декомпиляцию и дизассемблирование.** Вы имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (юлько в той мере, которая несовместима) явно разрешается соответствующим законом, несмотря на это ограничение

• **Разделение компонентов.** Данный Программный Продукт лицензируется как единый продукт. Его компоненты нельзя отделять друг от друга для использования более чем на одном компьютере

• **Аренда.** Данный Программный Продукт нельзя сдавать в прокат, передавая во временное пользование или уступать для использования в иных целях

• **Услуги по технической поддержке.** Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуги регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный код, предоставленный в рамках Услуги следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуги, то Microsoft может задействовать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас

• **Передача прав на программное обеспечение.** Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передалите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые обновления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения

• **Прекращение действия Соглашения.** Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программного Продукта вместе со всеми его компонентами

3. **АВТОРСКОЕ ПРАВО.** Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав программного Продукта и любые его копии принадлежат корпорации Microsoft или ее поставщикам Программного

Продукта охраняются законодательством об авторских правах и положениями международных договоров. Таким образом, Вы должны обращаться с данным Программным Продуктом, как любым другим материалом, охраняемым авторскими правами, **с тем исключением**, что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается

## ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТ С СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ, В ТОМ ЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СООТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВСЕГДА РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС. НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT, ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ, БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ, НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ. ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ НА НЕ КАСАТЬСЯ

## РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции той государственной территории, которой используется Программный Продукт. Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу Microsoft Sales Information Center, One Microsoft Way, Redmond, WA 98052-6399



Э. Джонс, Дж. Оланд  
Программирование в сетях Microsoft Windows.  
Мастер-класс

Перевод с английского под общей редакцией *А. В. Иванова*

Редактор *Н. Е. Субботина*

Предметный указатель *С. В. Дергачев*

Технический редактор *Н. Г. Тимченко*

Компьютерная верстка *Д. В. Петухов*

Дизайнер обложки *Е. В. Козлова*

Оригинал-макет выполнен с использованием  
издательской системы Adobe PageMaker 6.0

**TypeMarketFontLibrary**

легальный пользователь

ПОЛЬЗОВАТЕЛЬ

IN LEGAL USE

Совместное издание издательства <Русская Редакция>  
и издательства <Питер>

**MLРУССКАЯ РЕДАКЦИЯ С&П ПТЕР\***

Лицензия ЛР № 066422 от 19.03.99

Лицензия ИД № 01940 от 05.06.00

Подписано в печать 07.01.01 Формат 70х100/16 Физ. п. л. 38

Тираж 5000 экз. Заказ № 2043

ЗАО «Питер Бук»

196105, Санкт-Петербург, Благодатная ул., д. 67

Налоговая льгота - общероссийский классификатор продукции

ОК 005-93, том 2, 953000 - книги и брошюры

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького

Министерства РФ по делам печати, телерадиовещания

и средств массовых коммуникаций

197110, Санкт-Петербург, Чкаловский пр., 15