Sven B. Schreiber

Undocumented Windows 2000 secrets

A programming cookbook

Addison-Wesley
An imprint of Addison Wesley Longman, Inc.
Reading, Masachusetts • Harlow, England • Menlo Park, CaliforniaBerkley, California
• Don Mills, Ontario • SydneyBonn • Amsterdam • Tokyo • Mexico City

Свен Шрайбер

БИБЛИОТЕКА ПРОГРАММИСТА

Недокументированные возможности Windows 2000

Санкт-Петербург
Москва • Харьков • Минск
2002

ПИТЕР®

Свен Шрайбер

Недокументированные возможности Windows 2000
Библиотека программиста

Перевели с английского П. Анджан, А. Войтенко

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Пасечник*
Научный редактор	*А. Пасечник*
Художник	*Н. Биржаков*
Иллюстрации	*Р. Гришанов*
Корректоры	*М. Одинокова, И. Смирнова*
Верстка	*Р. Гришанов*

ББК 32.973-018.2
УДК 681.3.066

Шрайбер С.

Ш85 Недокументированные возможности Windows 2000. Библиотека программиста (+CD). — СПб.: Питер, 2002. — 544 с.: ил.

ISBN 5-318-00487-3

Данная книга предназначена для системных программистов, желающих максимально эффективно использовать возможности операционной системы, для которой они разрабатывают программное обеспечение. В книге содержится огромный объем информации, к которой можно применить атрибут «не документировано», кроме того, многие сведения ранее нигде не публиковались.

ISBN 5-318-00487-3
ISBN 0-201-72187-2 (англ.)

ЗАО «Питер Бук». 196105, Санкт-Петербург, Благодатная ул., д. 67.
Лицензия ИД № 01940 от 05.06.00.
Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 26.12.01. Формат 70×100$^{1}/_{16}$. Усл. п. л. 43,86. Тираж 5000 экз. Заказ № 2494.
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.

Краткое содержание

Содержание

Всем тем людям, которые никогда не прекращают спрашивать: «почему?»...

Предисловие

После завершения работы над моей первой книгой «Developing LDAP и ADSI Clients for Microsoft Exchange» (Разработка клиентов LDAP и ADSI для Microsoft Echange) в октябре 1999 года я был искренне убежден в том, что никогда в жизни больше не буду писать никаких книг. Однако, как красноречиво свидетельствуют строки, которые вы сейчас читаете, я находился в таком настроении недолго. На самом деле первые мысли о написании новой книги появились в моей голове в ноябре 1999 года, когда я экспериментировал с наиболее свежей на тот момент бета-версией Microsoft Windows 2000. Изучая ядро, его интерфейсы и структуры данных, я с удовольствием обнаружил, что, несмотря на ужасное имя, вызывающее у меня неприятные ассоциации с Windows 95 и 98, новая операционная система Windows 2000 мало чем отличается от старой доброй Windows NT.

Копаться в бинарном коде операционных систем всегда было моим любимым времяпрепровождением. За пару недель до того, как я принял решение написать эту книгу, в известном программистском журнале «Dr. Dobb'd Journal» (no 305, ноябрь 1999) была опубликована моя статья «Inside Windows NT System Data», в которой рассказывалось, как при помощи недокументированной функции ядра NtQuerySystemInformation() можно получить доступ к внутренним структурам данных операционной системы. Работая над статьей, я выполнил огромный объем исследовательских работ, в результате у меня скопилась масса материала, оставшегося неопубликованным. И тут я подумал: «Хей! Почему бы не написать книгу о внутреннем строении Windows 2000?» Идея была замечательная, особенно если принять во внимание очевидную схожесть Windows NT 4.0 и Windows 2000, а также тот огромный объем нигде недокументированных сведений о внутреннем строении этого семейства ОС, которыми я уже располагал и которые были слишком интересными для того, чтобы оставить их неопубликованными. И сейчас я не без гордости отмечаю, что моя идея была реализована в форме книги, которую вы держите сейчас в руках. Оформляя собранные мною сведения в форму, приемле-

мую для прочтения другими людьми, я открыл для себя множество новых интересных вещей, поэтому в данную книгу включен большой объем материала, который я не планировал включать в нее ранее.

Издательство «Addison-Wasley» уже давно занимается публикацией книг подобного рода. Перечень аналогичных материалов, опубликованных этим издательством, включает в себя такие известные книги, как двухтомная печатная версия классического списка прерываний PC, собранного Брауном и Джимом Кайлом (Brown R., and Kyle J. «PC Interrupts: A Programmer's Reference to BIOS, DOS, and Third-Party Calls». Reading, MA: Addison-Wasley, 1991, 1994, можно получить в электронном виде по адресу http://www.cs. cmu.edu/afs/cs.cmu.edu/user/ralf/pub/WWW/files.html), две редакции книги «Undocumented DOS» (Schulman, A. et al., «Undocumented DOS: A Programmer's Guide to Reserved MS-DOS Functions and Data Structures». Reading, MA: Addison-Wasley, 1990, 1992), продолжение этой серии, описывающее Windows 3.1 и названное «Undocumented Windows» (Schulman, A. et al., 1992), книга о внутреннем строении Windows Мэта Питрека (Matt Pietrek «Windows Internals»), а также «DOS Internals» Геофа Чаппела (Geoff Chappell) и «Dissecting DOS» Майкла Поданоффского (Podanoffsky, 1994). В прошлом многие другие авторы часто публиковали материалы подобного рода в других областях, например в области программирования аппаратного обеспечения. Эндрю Шульман и Мэтт Питрек написали также книги о недокументированных особенностях Windows 95. Однако, к великому сожалению, за последние годы не появилось ни одной подобной книги, ориентированной на аудиторию системных программистов Windows NT. К счастью, кое-какие сведения время от времени публиковались в форме статей в компьютерных журналах. В частности, Мэтт Питрек заполнил кое-какие пробелы в этой области в своей серии статей «Under the Hood», опубликованной в журнале «Microsoft Systems Journal» в 1996 году (недавно журнал «MSJ» сменил имя и стал называться «MSDN Magazine»). Емким источником информации о внутренностях NT является также журнал «NT Insider», ежемесячно издаваемый организацией «Open Systems Resources» (OSR). Бесплатную подписку на этот журнал можно оформить по адресу http://www.osr.com/publications_ntinsider.htm. И конечно же, нельзя не упомянуть о невероятном web-узле SysInternals (www.sysinternals.com), который принадлежит Марку Руссиновичу (Mark Russinovich) и Брюсу Когсвелу (Bryce Cogswell). Здесь можно обнаружить огромное количество мощных утилит (к некоторым из них прилагается исходный код), каждую из которых следовало бы включить в состав комплекта добавлений Windows NT Resource Kit.

Длительное время единственным источником информации о внутренностях операционной системы Windows NT являлся пакет разработчика драйверов Microsoft Windows NT Device Driver Kit (DDK). Однако документацию DDK сложно читать — многие люди находят ее непоследовательной, а подчас совершенно бесполезной. Мало того, документация DDK описывает лишь небольшую часть системных интерфейсов и структур данных. Подавляющее число механизмов ядра Windows упоминаются в этой документации как внутренние и недокументируемые. Таким образом разрабатывая любую нетривиальную программу, работающую на уровне ядра в операционной среде Windows NT, вам не обойтись без каких-либо

дополнительных источников информации. В прошлом раздобыть полезные сведения о ядре NT, равно как и связанный с этим исходный код, было непростой задачей. В 1997 году, с появлением книг «The Windows NT Device Driver Book» Арта Бекера (Art Baker) и «Windows NT File System Internals» Раджива Нагара (Rajeev Nagar) ситуация несколько улучшилась. В 1998 году появилась двухтомная версия «Inside Windows NT» и «Inside the Windows NT File System» изначально написанная Хелен Кастер (Helen Custer) и сильно переработанная Дэвидом Соломоном (David Solomon), который добавил в эти книги большой объем нового ранее не публиковавшегося материала. В начале 1999 года в свет вышла книга «Windows NT Device Driver Development», написанная экспертами разработки программ режима ядра Питером Дж. Вискарола (Peter G. Viscarola) и Энтони Мэйсоном (Anthony Mason). Оба автора являются консультантами Open Systems Resources (OSR), а организация OSR известна своими превосходными учебными курсами разработки драйверов режима ядра и программирования файловой системы. В том же году появилась книга «Developing Windows NT Device Drivers», написанная Эдвардом Н. Деккером (Edward N. Dekker) и Джозефом М. Ньюкомером (Joseph M. Newcomer). Обе эти книги можно считать первыми наиболее подробными и, что очень важно, *практически полезными* руководствами по программированию в режиме ядра в рабочей среде NT. Более обширный перечень книг, посвященных написанию драйверов для Windows NT, содержится в ноябрьском/декабрьском номере журнала «The NT Insider» за 1999 год (издается компанией Open Systems Resources).

В конце 1999 года к изучению глубин ядра Windows NT подключились некоторые новые авторы. Трио системных программистов из Индии, Прасад Дабак (Prasad Dabak), Сэндип Фадке (Sandeep Phadke) и Милинд Борате (Milind Borate), написали книгу «Undocumented Windows NT». Книга появилась на свет еще до появления финальной версии Windows 2000, поэтому в нее были включены сведения о Windows NT 3.51 и 4.0, а также о самой свежей на момент написания книги бета-версии Windows 2000. Наконец в январе 2000 года из печати вышла книга Гари Неббетта (Gary Nabbett) под названием «Windows NT/2000 Native API Reference». В этой книге содержится подробное, основательное, тщательно проработанное и содержащее огромное количество важнейших деталей описание всех функций естественного API, а также связанных с ними структур. Данная книга стала первой возможностью убедиться в правильности моих изысканий, связанных с функцией NtQuerySystemInformation(), об использовании которой я рассказал в журнале «Dr. Dobb's Journal». Мне было очень приятно узнать, что Наббет не только подтвердил правильность многих моих умозаключений, но и независимо от меня использовал точно такие же символьные имена для именования множества внутренних элементов ядра Windows. Было бы чрезвычайно полезно, если бы компания Microsoft опубликовала подобную книгу несколько лет назад!

Возможно, ознакомившись с кратким обзором литературы, вы придете к выводу, что на текущий момент о внутренностях Windows 2000 рассказано абсолютно все. Ничуть! Внутренние функции и структуры данных в недрах ядра Windows — это столь обширная область знаний, что для ее полного освещения двух-трех книг недостаточно. Книга, которую вы держите в руках, — это лишь один из множества

строительных камней, необходимых для формирования приемлемого набора знаний об архитектуре Windows 2000. Я надеюсь, что в ближайшее время появится большее количество публикаций на данную тему. Одной из последних публикаций, которую я бы хотел порекомендовать, является третья редакция книги «Inside Windows NT». В этой книге освещаются многие новые возможности, появившиеся в Windows 2000, поэтому новая версия книги называется теперь «Inside Windows 2000» (Solomon, Russinovich, 2000). Эту книгу можно считать выдающимся представителем серии «Inside Windows NT», так как в данном случае Марк Руссинович (Mark Russinovich) выступает в качестве соавтора Дэвида Соломона (David Solomon). Это обстоятельство указывает на то, что книгу можно покупать не глядя. Хочу также отметить, что между книгой, которую вы держите сейчас в руках, и книгой «Inside Windows 2000» нет существенных пересечений, поэтому я рекомендую вам приобрести обе.

Темы, затронутые в данной книге

Сравнивая данную книгу с ее предшественницами, отмечу, что она написана в лучших традициях старой доброй серии «Undocumented DOS» Шульмана (Schulman). Я до сих пор с любовью храню книги этой серии, так как они представляют собой идеальное соотношение между широтой рассматриваемых вопросов и глубиной проработки наиболее важных тем. На мой взгляд, фактически невозможно вместить в рамках одной книги всеобъемлющую документацию по сложной операционной системе и при этом обеспечить должный уровень детализации. Если в ваши планы не входит создание многотомной энциклопедии, вы либо выпускаете справочник (например, как у Неббетта), либо концентрируетесь на рассмотрении некоторых наиболее интересных вопросов, как поступили Шульман и его друзья. Книгу Неббетта можно считать всеобъемлющим справочником по функциям Native API. Сложно найти сведения, хотя бы кратко не освещенные в этой книге. Напротив, углубленное изучение и тщательное документирование отдельных специальных особенностей и внутренних механизмов Windows NT/2000 все еще остается непочатым краем для деятельности.

Как и «Undocumented DOS», данная книга рассказывает о недокументированных особенностях Windows 2000, которые показались мне как интересными, так и полезными. Следует отметить, что, представляя на суд общественности что-либо «недокументированное», всегда рискуешь оказаться в положении художника, который занимается «искусством ради искусства». Раскрытие недокументированных возможностей операционной системы может представлять огромный интерес для человека, который занимается исследованиями этих возможностей, вместе с тем вовсе не обязательно, что раскрываемые им недокументируемые возможности могут быть использованы на практике другими людьми. Далеко не все, что является недокументированным, автоматически является полезным для разработчиков программного обеспечения. Многие внутренние механизмы операционной системы являются внутренними в самом строгом смысле этого слова, а это означает, что использование этих механизмов при разработке собственных программ не рекомендуется. Все же

огромное количество внутренних элементов операционных систем Microsoft можно и должно изучать. Компания Microsoft печально известна тем, что стремится любыми средствами предотвратить распространение сведений о внутреннем строении своих ОС среди сторонних разработчиков. Точка зрения Microsoft сводится к тому, что разработчики не должны знать ничего лишнего об ОС, в которой они работают. Подчас такая позиция наносит огромный вред и снижает эффективность разрабатываемых программ. Моим любимым примером на эту тему является оригинальный и чрезвычайно удобный интерфейс MS-DOS Network Redirector, описанный в главе 8 второго издания «Undocumented DOS» Шульмана (Schoulman A., Brown R., Maxey D., Michels R. J., Kyle J. «Undocumented DOS: 2nd ed. A Programmer's Guide to Reserved MS-DOS Functions and Data Structures». Reading, MA: Addison-Wesley, 1992). Если бы компания Microsoft документировала этот замечательный интерфейс в момент, когда он был добавлен в состав ОС, множество сторонних разработчиков по всему миру смогли бы сэкономить массу времени и усилий, а разрабатываемые ими продукты были бы более эффективными и содержащими меньшее количество ошибок. К сожалению, подобная политика ненужной секретности в отношении многих, подчас весьма полезных внутренних механизмов ОС компании Microsoft была продолжена с появлением Windows 3.x, Windows 9x, Windows NT и Windows 2000. К счастью, информацию, которая скрывается от нас компанией Microsoft, можно найти в книгах, подобных той, которую вы держите сейчас в руках.

Познакомив вас с базовой архитектурой Windows 2000 и сопроводив вас подробными инструкциями о том, как превратить ваш компьютер в рабочую станцию отладки ядра Windows 2000, я расскажу вам о некоторых весьма интересных уголках внутри ядра этой операционной системы. Как правило, в начале каждой главы содержатся теоретические сведения, связанные с обсуждаемой темой. После этого в главе приводится исходный код, иллюстрирующий работу с соответствующими механизмами Windows 2000. Все примеры написаны на обыкновенном C. Я надеюсь, что все читатели данной книги хорошо знакомы с C. В среде Windows 2000 этот язык является одним из стандартных языков программирования, поэтому он поддерживается всеми основными средствами разработки программ для Windows 2000.

В разных главах книги рассматриваются различные составляющие ядра, однако я умышленно не включил в книгу всеобъемлющее описание архитектуры ядра Windows 2000. Если вы желаете ознакомиться с таким описанием, обратитесь к книге «Inside Windows 2000» Соломона и Руссиновича. В ней вы найдете обобщенное и теоретизированное описание внутренностей Windows 2000. Хочу обратить ваше внимание на то, что ни в «Inside Windows 2000» Соломона и Руссиновича, ни в «Windows NT/2000 Native API Reference» Неббетта вы не найдете практических примеров исходного кода, равно как и полноценных прикладных программ, которые взаимодействуют с функционирующей системой и выполняют операции на уровне ядра Windows 2000. В отличие от двух вышеупомянутых книг данная книга изобилует примерами исходного кода, иллюстрирующего обсуждаемые концепции и возможности Windows 2000. Весь обсуждаемый исходный код, равно как и готовые к запуску прикладные программы, содержатся на прилагаемом к книге компакт-диске. Таким образом, вы можете использовать этот код в своих собствен-

ных приложениях, дополнить его своими собственными фрагментами или применять в том виде, в котором он есть. В книге рассматриваются следующие основные темы:

- использование отладочных интерфейсов Windows 2000;
- загрузка, грамматический разбор и использование файлов символьных идентификаторов ядра Windows 2000;
- внутреннее строение формата Microsoft PDB;
- взаимодействие с интерфейсом Native API;
- создание простых драйверов режима ядра;
- изучение системной памяти Windows 2000;
- перехват вызовов к интерфейсу Native API и слежение за обращениями к этому интерфейсу;
- обращение к функциям режима ядра из приложений пользовательского режима;
- обращение к неэкспортируемым функциям ядра;
- изучение вселенной объектов ядра Windows 2000.

Читая книгу, вы узнаете много интересного помимо перечисленных здесь основных тезисов. Когда я писал книгу, то старался довести до сведения читателей абсолютно все, что мне известно о том или ином обсуждаемом вопросе.

Внутренняя организация книги

Определяя последовательность, в которой следовало бы расположить материал, я решил организовать книгу в виде семи глав, расположенных в порядке, который был бы удобен для начинающих системных программистов Windows 2000. Исходя из этого я старался размещать в начале главы описание любых новых концепций и технологий, чтобы облегчить дальнейшее знакомство с материалом. Таким образом, на мой взгляд, новички в мире системного программирования Windows 2000 должны читать главы книги по порядку, в той последовательности, в которой эти главы расположены в книге. Предлагаемый подход может оказаться чересчур утомительным для настоящих экспертов, которые в общих чертах знакомы с теорией и приобрели книгу прежде всего для получения действительно недокументированных и ранее нигде не опубликованных сведений. Однако для любого эксперта не составит труда пропустить материал, с которым он хорошо знаком.

Итак в главах книги вы найдете обсуждение следующих вопросов:

- **Глава 1** начинается с инструкций, руководствуясь которыми вы сможете установить, настроить и использовать отладчик ядра Windows 2000 Kernel Debugger. Отладчик ядра является чрезвычайно важным инструментом исследования внутренностей системы, без которого не может обойтись ни один настоящий хакер ядра. Помимо вопросов, связанных с установкой и использованием отладчика, в главе обсуждаются официальные отладочные интерфейсы Windows 2000. Эти интерфейсы входят в состав ОС в виде компонент psapi.dll, imagehlp.dll и

dbghelp.dll. В завершающей части главы рассматриваются файловые форматы Microsoft Code View и Program Database (PDB). Здесь же приводится описание простого компонента грамматического разбора символьных файлов и прилагаемого к нему клиентского приложения.

- **Глава 2** знакомит читателя с интерфейсом Windows 2000 Native API. Кроме того, в этой главе описывается основной механизм синхронизации системных служб, различные группы функций API, экспортируемые модулями ntdll.dll и ntoskrnl.dll, а также типы данных, наиболее часто используемые этими компонентами.
- **Глава 3** — это краткое и упрощенное введение в разработку драйверов режима ядра. Ее ни в коем случае нельзя рассматривать как полноценное руководство для разработчиков аппаратных драйверов. В главе содержатся лишь те сведения о драйверах, которые являются важными для понимания исходного кода, обсуждаемого в последующих главах. Здесь вы найдете информацию о том, каким образом происходит загрузка и выгрузка драйверов в процессе функционирования ОС с использованием интерфейса Service Control Manager. Здесь же содержится описание мастера создания скелета драйвера, при помощи которого вы сможете автоматически создать проект драйвера в Visual C/C++. Исполняемый файл мастера с исходным кодом содержится на прилагаемом к книге компакт-диске.
- **Глава 4** наверное будет самой сложной для тех, кто боится всего, что связано с аппаратным обеспечением. Эта глава начинается с подробного описания механизмов управления памятью процессора Intel Pentium, которые используются диспетчером памяти Windows 2000. Те, кто осилит этот раздел главы, будут вознаграждены исходным кодом драйвера, позволяющего просматривать области памяти, доступ к которым закрыт для обычных приложений, а также внутренние структуры данных диспетчера памяти Windows 2000. Также в главе приводится карта памяти Windows 2000, иллюстрирующая, каким образом система использует обширное 4-Гбайтное адресное пространство, работу с которым поддерживают процессоры семейства Pentium.
- **Глава 5** подробно рассказывает о том, как можно перехватить обращение к функции Native API. Основное внимание уделяется слежению за передаваемыми в функцию аргументами, а также наблюдению за обращениями к файлам и реестру. В данной главе активно используется ассемблер, а кроме того, применяются манипуляции со стеком ЦПУ.
- **Глава 6** предлагает эффективное решение проблемы, которую в мире Windows 2000 принято считать нерешаемой, а именно обращение к коду режима ядра из приложения пользовательского режима. Рассматриваемый в данной главе исходный код является связующим звеном между подсистемой Win32 и интерфейсами ядра внутри модулей ntoskrnl.exe, hal.dll и других базовых компонент Windows 2000. В главе рассказывается также о том, каким образом можно обратиться фактически к любой, даже неэкспортируемой функции ядра. Для обнаружения точек входа неэкспортируемых функций предлагается использовать файлы символьных идентификаторов Windows 2000.

- **Глава 7** содержит описание диспетчера объектов Windows 2000. Внутренняя структура объектов ядра — один из наиболее тщательно охраняемых секретов Microsoft. Во всех предлагаемых Microsoft исходных файлах ссылки на любые объекты ядра, как правило, обозначаются безликим указателем void*. В данной главе рассказывается о том, на что именно указывает данный указатель, а также дается представление о том, каким образом система работает со структурами и дескрипторами объектов. Подробно рассматривается внутреннее устройство объектов процесса и потока. В завершающей части главы рассматривается исходный код приложения, которое отображает на экране иерархическую структуру объектов ядра. Для этого выполняется анализ разнообразных недокументированных системных структур.
- **Приложение А** напрямую связано с главой 1 и содержит все команды и параметры отладчика Windows 2000 Kernel Debugger.
- **Приложение Б** напрямую связано с главой 2 и содержит описание функций API, экспортируемых модулями ядра Windows 2000.
- **Приложение В** содержит обширную коллекцию констант и типов данных Windows 2000, перечисленных в алфавитном порядке. В данном перечне документируются многие недокументированные структуры ядра Windows 2000, которые рассматриваются и используются при обсуждении остального материала книги.

Можно заметить, что в книге содержится огромный объем информации, к которой можно применить атрибут «не документировано», кроме того, многие сведения ранее нигде не публиковались.

Для кого предназначена эта книга

Данная книга предназначена системным программистам, желающим максимально эффективно использовать возможности операционной системы, для которой они разрабатывают программное обеспечение. Предупреждение номер один: если вы разрабатываете программы, предназначенные в основном для работы в среде Windows 95, 98 или Me (Millennium Edition), можете отложить эту книгу в сторону. Архитектура Windows 9x/Me сильно отличается от архитектуры Windows NT/2000, поэтому при разработке программ для Windows 9x/Me вы не сможете применить методы, рассматриваемые в данной книге. Предупреждение номер два: в этой книге отсутствует какая-либо информация о процессорах семейства Alpha или многопроцессорных системах — целью моих исследований являются однопроцессорные системы на базе 32-битных процессоров семейства Intel i386. Предупреждение номер три: книга написана не для слабонервных. Читая ее, вы столкнетесь с технологиями и методами, о которых среднестатистический программист Win32 ничего не знает. Ядро Windows 2000 — это совершенно другой мир, который фактически ничем не напоминает подсистему Win32, построенную на его основе. Некоторые методики взаимодействия с ядром, описанные ближе к концу книги, окажутся новыми даже для опытных программистов режима ядра. Будьте уверены в том, что учителя высшей школы, равно как и представители Microsoft, всегда будут предостерегать вас от использования технологий и приемов, описанных здесь.

Если, несмотря на предостережения, вы все же продолжаете читать эти строки, — значит вы свободный от предрассудков смелый исследователь, который желает знать все о вещах, скрытых под поверхностью операционной системы Windows 2000. Отлично! Даже если в своей повседневной работе вы не собираетесь постоянно использовать приемы, описанные в данной книге, все равно, прочтя ее вы только выиграете. Знание принципов работы внутренностей Windows 2000, скрытых под поверхностью интерфейса прикладных программ, никогда не будет лишним. Подобные сведения могут оказаться полезными в процессе отладки и оптимизации приложений, а также помогут вам избежать нежелательных побочных эффектов, проявление которых, как правило, связано с отсутствием представления о работе внутренних механизмов системы.

Какими знаниями должен обладать читатель книги? Я ожидаю, что моя аудитория свободно владеет C, а также обладает базовыми знаниями основ программирования в среде Win32. Если вы обладаете опытом разработки драйверов режима ядра, — значит, читая книгу, вы будете чувствовать себя более комфортно, однако для того, чтобы хорошо освоить материал, вовсе не обязательно быть опытным разработчиком драйверов. Базовые сведения о том, как разработать собственный драйвер, содержатся в главе 3 данной книги. Там вы найдете все необходимое для того, чтобы продолжить знакомство с глубинами ядра Windows 2000. Однако следует учитывать, что это краткое введение в разработку драйверов режима ядра вовсе не следует рассматривать как подробное руководство. Если вы заинтересованы в разработке драйверов режима ядра, советую вам приобрести одну из посвященных этому вопросу книг (например, Viscarola P. G. Mason W. A. «Windows NT Device Driver Development». Indianapolis, IN: Macmillan Technical Publishing 1999 или Dekker N. E., Newcomer, J. M. «Developing Windows NT Device Drivers: A Programmer's Handbook». Reading, MA: Addisson-Wesley, 1999).

В некоторых главах книги активно используется язык ассемблера (ASM). Фрагменты, написанные на ассемблере, включаются в текст программ C. Читатель вовсе не обязан быть экспертом в области ассемблера, однако базовые знания этого языка могут оказаться полезными при освоении некоторых разделов книги. Если ранее вы никогда не занимались изучением ассемблера, возможно, такие разделы окажутся для вас сложными. Вполне вероятно, вы решите вообще пропустить связанные с ассемблером места книги. Однако я рекомендую вам прочитать их хотя бы частично. Вы можете ознакомиться с основным материалом главы и пропустить те фрагменты, где обсуждается ассемблерный код. Фрагменты ассемблерного кода всегда включаются в функции C, являющиеся, с точки зрения программиста, удобными оболочками, поэтому вы можете не тратить время на изучение внутреннего строения этих функций, а просто использовать их на практике.

Соглашения, используемые в данной книге

Основной темой обсуждения в данной книге является операционная система Windows 2000, однако вы можете обнаружить, что большая часть сведений применима также в отношении Windows NT 4.0 — большая часть приложений, содержащихся на прилагаемом к книге компакт-диске, отлично работает в среде NT 4.0.

В своей книге я не рассматриваю Windows NT 3.x. В настоящее время операционная система NT 3.51 уже устарела, однако, несмотря на это, данная ОС из семейства NT является моей любимой, так как она обладает относительно небольшим размером и работает достаточно быстро из-за того, что не тратит слишком много процессорного времени на обслуживание своего графического интерфейса. Что касается Windows NT 4.0, я сделал все, что мог, для того чтобы обеспечить совместимость моих программ с этой ОС, так как эта ОС все еще активно используется. Многие компании используют сети, основанные на классической концепции доменов Windows NT, и потребуется значительное время для того, чтобы перейти на использование технологии Active Directory, которая появилась в Windows 2000. В некоторых случаях обеспечить выполнение одного и того же кода в обеих операционных системах невозможно. Это связано с тем, что строение многих внутренних структур данных в этих ОС отличается. По этой причине в некоторых разделах исходного кода содержится проверка версии ОС и в зависимости от этого происходит выполнение разных участков кода.

Что касается терминологии, то хочу отметить, что когда я упоминаю Windows 2000, я, как правило, имею в виду также и Windows NT 4.0. Не стоит забывать о том, что новое непривычное имя — это всего лишь хитрый маркетинговый прием, направленный на то, чтобы стимулировать продажи операционной системы, которая должна была бы называться Windows NT 5.0. Хочу также отметить, что термин Windows NT без указания версии относится к платформе NT в целом, включая Windows 2000 и Windows NT 4.0. Обратите внимание, что Windows NT 3.x не входит в этот список. Рассказывая об особенностях, имеющих отношение к той или иной версии ОС, я использую термины Windows 2000 и Windows NT 4.0, так чтобы было понятно, о какой именно версии ОС идет речь.

Исходный код на компакт-диске

Исходный код, записанный на компакт-диск и частично воспроизведенный на страницах данной книги, написан для Microsoft Visual C/C++ 6.0 с установленным пакетом добавлений Service Pack 3. Если вы намерены заново откомпилировать или модифицировать какую-либо из рассматриваемых здесь программ, вы должны установить также последние версии SDK (Software Development Kit) и DDK (Device Driver Kit). Данные комплекты содержат последние версии заголовочных файлов и импортируемых библиотек. Убедитесь в том, что пути поиска заголовочных и библиотечных файлов настроены таким образом, чтобы компилятор и компоновщик прежде всего обращались бы к файлам, входящим в комплекты SDK и DDK, и только после этого — к файлам, устанавливаемым в составе Visual C/C++. Оба комплекта распространяются на компакт-дисках или DVD-дисках как часть подписки Microsoft Developer Network (MSDN) Professional и MSDN Universal. Если вы еще не подписались на MSDN, сделайте это! Вы будете получать последние обновления всех операционных систем и средств разработки, распространяемых компанией Microsoft, а также более одного Гигабайта ценнейшей технической документации. Подписка на MSDN стоит денег, однако я полагаю, что связанные с этим затраты с лихвой окупят себя. Более подробно о подписке на MSDN можно узнать по адресу http://msdn.microsoft.com/subscriptions/prodinfo/overview.asp.

На прилагаемом к книге компакт-диске содержится исходный код всех рассматриваемых в ней программ, а также откомпилированные и скомпонованные, а следовательно, готовые к использованию исполняемые файлы. Структура каталогов соответствует стандарту Visual Studio 6.0: для каждого модуля существует каталог, содержащий исходные файлы (код на языке C, заголовочные файлы, сценарии ресурсов, файлы определений, информация о проекте и рабочем пространстве и т. п.). Структура каталогов компакт-диска отражена на рис. П.1. Все исходные файлы и файлы проектов располагаются в каталоге \src. Для каждого отдельного проекта в каталоге содержится отдельный подкаталог. В подкаталоге common содержатся файлы, общие для всех проектов. Каталог \bin содержит все исполняемые файлы рассматриваемых программ. Здесь располагаются все файлы типа EXE, DLL и SYS, необходимые для запуска программ, рассматриваемых в книге. Благодаря этому вы можете запускать эти программы прямо с компакт-диска. В каталоге \tools располагается коллекция утилит, разработанных сторонними производителями. Я полагаю, что эти утилиты окажутся чрезвычайно полезными для читателей, которые увлекаются вопросами, рассматриваемыми в данной книге.

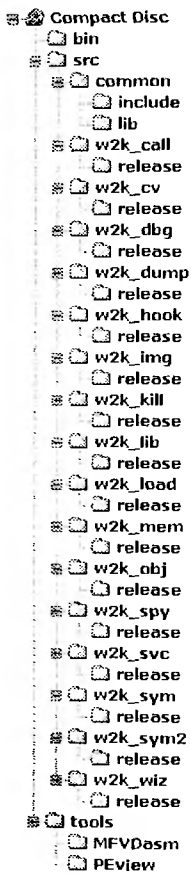

Рис. П.1. Структура каталогов прилагаемого к книге компакт-диска

Чтобы заново откомпилировать одну из программ, просто скопируйте базовый каталог модуля, включая подкаталог release, в папку, в которой располагаются ваши собственные проекты Visual C/C++. В каждом базовом каталоге расположены файлы типа DSW и DSP, которые являются файлами рабочего пространства и проекта и содержат служебную информацию о проекте, используемую средой разработки Visual Studio. Чтобы заново откомпилировать проект, необходимо выполнить следующее: работая с Visual Studio, откройте DSW-файл, выберите активную конфигурацию (например, Release) и в главном меню выберите Build (компоновка) ▸ Build (компоновка) или Build (компоновка) ▸ Rebuild All (заново скомпоновать). Обратите внимание на то, что некоторые заголовочные файлы содержат дополнительные инструкции компоновщику в форме директив #pragma. Благодаря этому приему вы получаете возможность откомпилировать все содержащиеся на компакт-диске примеры с использованием конфигурации Visual Studio по умолчанию. Другими словами, для того чтобы получить корректный исполняемый файл, вам не потребуется перенастраивать конфигурацию проекта.

Большая часть кода совершенно несовместима с Windows 9x, по этой причине я решил отказаться от поддержки символов ANSI. В рабочей среде Windows 2000 основным стандартом кодирования символов является Unicode, который предусматривает кодирование каждого символа 16-битным числом. По этой причине строки определяются как массивы значений WORD или как указатели на последовательность значений WORD. Однако следует учитывать, что в некоторых очень редких случаях система все же использует ANSI. Отказ от поддержки ANSI сильно упрощает дело, однако все же любой хороший программист Win32 должен обеспечить в разрабатываемой им программе поддержку обоих стандартов, как ANSI, так и Unicode. В этом случае появляется возможность запуска разрабатываемой программы в среде устаревших операционных систем Windows 95 и 98. К коду, рассматриваемому в данной книге, данное замечание не относится, так как большая часть этого кода работает ниже уровня подсистемы Win32. Благодаря этому появляется возможность в полной мере использовать естественные для Windows 2000 внутренние механизмы. Исключением является рассматриваемая в книге и записанная на прилагаемом компакт-диске библиотека w2k_img.dll, а также связанное с этой библиотекой приложение w2k_sym2.exe. Оба этих проекта рассматриваются ближе к концу первой главы. Эта библиотека DLL поддерживает как Unicode, так и ANSI, а приложение откомпилировано в стандарте ANSI. Из всех программ, записанных на компакт-диске, только это приложение можно запустить в среде Windows 95.

Между проектами, записанными на компакт-диске, существуют множественные зависимости. В частности, драйвер режима ядра w2k_spy.sys, а также библиотека w2k_lib.dll активно используются многими приложениями, обсуждаемыми в разных главах книги. Зависимости между записанными на компакт-диске приложениями и модулями проиллюстрированы на рис. П.2. Приведенную на этом рисунке диаграмму следует читать слева направо. Например, приложение w2k_obj.exe импортирует функции из модулей w2k_call.dll и w2k_lib.dll, а библиотека w2k_call.dll, в свою очередь, использует код модулей w2k_lib.dll и w2k_img.dll, кроме того, выполняются вызовы Device I/O Control, обслуживаемые драйвером w2k_spy.sys. Это

означает, что для того, чтобы корректно запустить приложение w2k_obj.exe, все файлы, используемые этим приложением, следует разместить в том же каталоге, в котором размещается и сам исполняемый файл w2k_obj.exe. Обратите внимание, что в состав диаграммы входят также компоненты imagehlp.dll и psapi.dll, которые на самом деле являются компонентами Microsoft. Это сделано для того, чтобы отметить, что эти компоненты используются библиотекой w2k_dbg.dll, в то время как остальные модули, записанные на компакт-диске, в них не нуждаются.

Об одном из записанных на компакт-диске модулей необходимо рассказать подробнее. Я имею в виду прикладную библиотеку Windows 2000 Utility Library, проект которой расположен в каталоге w2k_lib. Прежде всего хочу отметить, что в состав этой библиотеки входит огромное количество функций, рассматриваемых в разных главах книги. Но помимо этих функций в данной библиотеке также содержится большой объем кода Win32, который не связан напрямую с рассматриваемыми в книге вопросами, однако с успехом может быть использован в ваших собственных проектах. Здесь вы найдете функции работы с памятью, реестром, объектами и связанными списками, подпрограммы вычисления CRC32, генерации псевдослучайных чисел, код проверки версий операционной системы и различных файлов, а также большой объем другого чрезвычайно полезного кода. Огромный по размерам файл w2k_lib.c является вместительным хранилищем самого разнообразного полезного кода, который я написал сам для себя за последние несколько лет. Я надеюсь, что мои разработки окажутся полезными для многих читателей этой книги и существенно упростят жизнь многих программистов Win32.

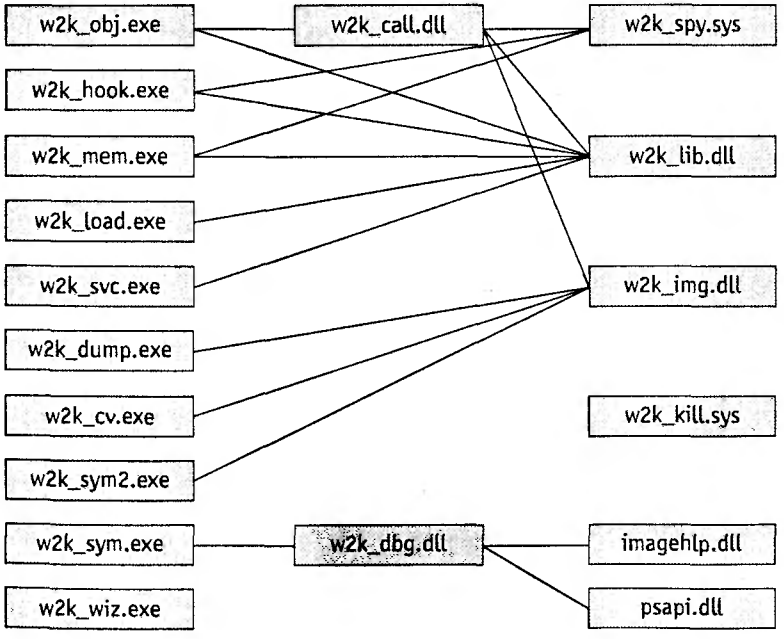

Рис. П.2. Взаимозависимости между программами, записанными на компакт-диске

Записанные на компакт-диске утилиты сторонних разработчиков

Помимо программ, которые я написал специально для данной книги, на прилагаемом к книге компакт-диске содержатся очень полезные программные средства, разработанные другими людьми. Я чрезвычайно признателен Джину-Луису Сигне (Jean-Louis Seigne) и Вэйну Дж. Рэдберну (Wayne J. Radburn), которые предоставили мне право опубликовать разработанные ими программные средства на компакт-диске, прилагаемом к данной книге. В каталоге \tools вы обнаружите следующие полезные утилиты:

- *The Multi-Format Visual Disassembler* (MFVDasm) разработан Джином-Луисом Сигне, занимающимся разработкой программного обеспечения для Windows с 1990 года. На самом деле MFVDasm — это не только мощный и удобный дизассемблер, но и анализатор файлов формата PE (Portable Executable), утилита просмотра шестнадцатеричных дампов, а также программа просмотра ASM-кода. И все это сосредоточено в одном программном продукте. В каталоге \tools\MFVDasm компакт-диска содержится полнофункциональная демо-версия, срок действия которой ограничен. Программа защищена при помощи программного средства Softlocx, разработанного компанией BitArts. Чтобы приобрести полноценную коммерческую версию (срок действия которой не ограничен), необходимо при помощи кредитной карточки заплатить Джину-Луису Сигне 100 американских долларов. Последнюю версию программы MFVDasm можно получить по адресу http://redirect.to/MFVDasm. Запросы на приобретение программы, а также любые связанные с ней вопросы можно направлять по адресу MFVDasm@redirect.to.
- *PE and COFF File Viewer* (PEview) разработан Вэйном Дж. Рэдберном и распространяется абсолютно бесплатно в качестве специального дополнения к данной книге. Как и я, Вэйн является закоренелым любителем программирования на ассемблере для Win32. Разница между нами состоит в том, что он до сих пор продолжает этим заниматься, в то время как я все же перешел на C. Программирование приложений Win32 на ассемблере — нелегкая задача, поэтому вы можете быть уверенными в том, что Вэйн — настоящий эксперт. Без сомнений, PEview является наиболее мощным известным мне средством анализа PE-файлов, и это действительно незаменимый инструмент для всех, занимающихся изучением внутреннего устройства операционных систем. Утилита PEview обеспечивает простой и быстрый способ просмотра структуры и содержимого файлов, записанных в форматах 32-bit Portable Executable (PE) и Component Object File Format (COFF), и позволяет просматривать файлы типа EXE, DLL, OBJ, LIB, DBG, а также некоторых других форматов. Вы можете посетить домашнюю страничку Вейна по адресу http://www.magma.ca/~wir/ или послать ему письмо по адресу wir@magma.ca.

Упомянутые программы сторонних разработчиков распространяются в соответствии с условиями специальных лицензионных соглашений с соответствующими авторами, поэтому ни издатель книги, ни я не несем никакой ответственности, связанной с использованием этих программ. Более подробно об условиях использования этих программных средств можно узнать из информации, которую они отображают при запуске.

Те, без кого не состоялась бы эта книга

До того как я написал свою первую книгу, я не думал, что над каждой издаваемой книгой работает столько людей. Написание рукописи — лишь один из многочисленных этапов, которые необходимо выполнить, прежде чем книга появится на полках в магазинах. Данный раздел посвящается всем людям, которые принимали значительное участие в создании данной книги.

Прежде всего хочу поблагодарить Гэри Кларка (Gary Clarke), ранее работавшего в издательстве «Addison-Wesley» и приложившего усилия к тому, чтобы началась работа над данным проектом. К сожалению, Гэри покинул издательство «Addison-Wesley» незадолго до того, как я написал первые строки рукописи. Однако, к счастью, судьба снова свела меня с Карен Геттман (Karen Gettman) и Мэри Харт (Mary Hart), которые занимались координацией работы над рукописью моей первой книги. И это была большая удача, о лучших компаньонах я не мог даже и мечтать. Однако когда я дописывал последние абзацы главы 7, мое счастье несколько омрачилось, так как я узнал о том, что Мэри ушла из издательства «Addison-Wesley». К счастью, ее место заняла Эмили Фрей (Emily Frey). Я также хочу выразить благодарность Мамате Редди (Mamata Reddy), координировавшей производство книги, Курту Джонсону (Curt Johnson), Дженнифер Лавински (Jennifer Lawinski) и Чанда Д. Лири-Куту (Chanda D. Leary-Coutu) из маркетингового отдела, Кэти Ноэз (Katie Noyes), разработавшей обложку, а также Бес Хаес (Beth Hayes), занимавшейся редактированием рукописи. Еще несколько человек из издательства «Addisson-Wesley», чьи имена остались для меня неизвестными, принимали активное участие в производстве и продаже этой книги, и я хочу от чистого сердца поблагодарить их за это. Помимо этого, хочу сказать огромное спасибо королеве моего сердца Герде Б. Градл (Gerda B. Gradl), моим родителям Алле Шрейбер (Alla Schreiber) и Олафу Шрейбер (Olaf Schreiber), а также моей коллеге Рите Спрангер (Rita Spranger) за постоянную поддержку и ободрение, которые они оказывали мне в течение всей работы над книгой. Написание книги — это подчас уединенное занятие, и хорошо, когда время от времени рядом есть кто-то, с кем ты можешь поговорить о том, чем занимаешься.

И, наконец, огромное, огромное спасибо Роине Столту (Roine Stolt), лидеру шведской прогрессивной рок-группы «The Flower Kings» (http://www.users.wineasy.se/flowerkings/) за его невероятную и неописуемую музыку, которая была для меня неиссякаемым источником вдохновения за все время работы над рукописью. Роине постоянно напоминал мне о том, что «Все мы — звездная пыль» (название песни).

We believe in the light,
We believe in love,
Every precious little thing.
We believe you can still surrender,
You can serve the Flower King.

(Замечание: отрывок из песни «The Flower King», написанной Роине Столтом в 1994 году, использован с любезного разрешения автора.)

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства http://www.piter.com.

1 Механизмы отладки в среде Windows 2000

Большая часть сведений, содержащихся в данной книге, являются недокументированными, поэтому для того, чтобы получить их, требуется копаться в исполняемом коде операционной системы. В состав комплекта Windows 2000 Device Driver Kit (DDK) входит мощный отладчик, который позволяет решать подобные задачи. В начале данной главы содержатся подробные инструкции относительно того, как настроить полноценную среду отладки на вашей машине. Читая последующие главы, вы будете часто обращаться к помощи отладчика ядра (Kernel Debugger), который поможет вам в получении сведений о разнообразных внутренних механизмах Windows 2000. Если этот отладчик покажется вам неудобным или непривычным, вы можете попытаться использовать для отладки ядра другие, более привычные вам средства. Чтобы упростить эту задачу, в главе содержится описание документированных и недокументированных отладочных интерфейсов Windows 2000. Кроме того, в данной главе вы обнаружите сведения о файлах символьных идентификаторов Windows 2000. В главе рассматриваются две библиотеки, а также работающие с этими библиотеками прикладные программы, которые позволяют получить список процессов, перечень загруженных в память системных и прикладных модулей, а также разнообразную информацию о символьных идентификаторах, извлекаемую из символьных файлов Windows 2000. В качестве специального дополнения в конце главы содержится первое публичное описание нового файлового формата Microsoft Program Database (PDB).

Установка и настройка отладочной среды

«Эй! Но ведь я не собираюсь отлаживать программу Windows 2000! Прежде всего я хочу написать такую программу!» — воскликнут многие, прочитав заголовок раздела. Не беспокойтесь! В книге, которую вы держите сейчас в руках, речь пойдет прежде всего именно о разработке новых программ, а не об отладке существующих. Но зачем тогда нужно настраивать отладочную среду? Ответ прост: отладчик — это потайная дверь, используя которую можно получить доступ к самым удаленным уголкам операционной системы. Конечно же, изначально люди, занимавшиеся разработкой отладчика, не планировали, что это программное средство

будет использоваться для доступа к внутренностям ОС. Несмотря на это, любое хорошее средство отладки должно предоставить работающему с ним программисту какую-то, хотя бы минимально полезную, информацию о системе, в которой выполняется отлаживаемый код. Представьте, что разрабатываемая вами программа дала сбой. Согласитесь, что если при этом вы получите от отладчика только лишь восьмизначный адрес сбойной команды, идентифицирующий одну из ячеек огромного 4-Гбайтного пространства, этого будет явно недостаточно для того, чтобы установить причину сбоя. Хороший отладчик должен по крайней мере, сообщить, в каком программном модуле произошел сбой и, в идеале, имя функции, к которой обратилось приложение. Таким образом, отладчик должен обладать куда более обширными сведениями об ОС, чем те, что содержатся в руководствах пользователя. Именно этими возможностями отладчика можно с успехом воспользоваться для исследования внутреннего строения ОС.

В состав операционной системы Windows 2000 входят два стандартных отладчика: `WinDbg.exe` (произносится как «виндбаг») и `i386kb.exe`. `WinDbg.exe` является приложением Win32 с графическим интерфейсом, в то время как `i386kb.exe` — это консольное приложение. Я достаточно долго работал с обоими этими отладчиками и в конце концов пришел к выводу, что `i386kb.exe` лучше, так как этот отладчик обладает более широким набором возможностей. Однако в последнее время появились сведения о том, что новая версия `WinDbg.exe` существенно улучшена. Об этом написано в статье «There's a New WinDBG in Town — And It Doesn't Suck Anymore» (Появился новый WinDBG, и с ним действительно можно работать), опубликованной в майском и июньском номерах журнала «The NT Insider» за 2000 год (журнал «The NT Insider» издается компанией Open Systems Resources). Все же примеры, рассматриваемые в данной книге и так или иначе связанные с отладкой, относятся к отладчику `i386kb.exe`. Как можно предположить, аббревиатура `i386` обозначает процессорную платформу (в данном случае имеется в виду семейство процессоров Intel 386, включая процессоры Pentium), а буквы `kb` расшифровываются как *Kernel Debugger* — отладчик ядра. Отладчик Windows 2000 Kernel Debugger является чрезвычайно мощным инструментом. В частности, он может работать с файлами символьных идентификаторов, содержащимися на установочных комакт-дисках Windows 2000, благодаря чему вы можете получить сведения фактически о любом адресе в системной памяти. Помимо этого, отладчик ядра позволяет дизассемблировать бинарный код, выводить на экран шестнадцатеричные листинги содержимого памяти и даже показывать вам внутреннее строение некоторых внутренних структур ядра. Вся эта информация предоставляется отладчиком совершенно открыто — командный интерфейс отладчика хорошо документирован в электронной справочной системе.

Подготовка к получению полного содержимого памяти в момент сбоя системы

Все, что я до сих пор рассказывал вам об отладчике Windows 2000 Kernel Debugger, звучит неплохо, однако прежде, чем вы сможете приступить к полноценной работе с этим отладчиком, вы должны выполнить кое-какую подготовительную работу. Прежде всего следует отметить, что, как правило, в ходе сеанса отладки используются два компьютера, соединенных кабелем, — на одном из них работает отладчик,

в то время как на другом функционирует отлаживаемое приложение. Однако в некоторых случаях, когда не требуется осуществлять отладку «вживую», можно обойтись и без второго компьютера. В частности, если в ходе выполнения отлаживаемого приложения система отображает BSOD (Blue Screen Of Death — голубой экран смерти), вы можете записать содержимое оперативной памяти в момент возникновения сбоя в файл, затем перезапустить систему и проанализировать содержимое этого файла, чтобы установить причину сбоя. Этот метод часто называют отладкой *post mortem* (в переводе с латинского *post mortem* означает *после смерти*). Именно такая методика является основным способом исследования внутренностей Windows 2000, который применялся мной при написании данной книги. При этом основной задачей является исследование содержимого памяти, принадлежащей ядру операционной системы. В большинстве случаев не имеет значения, получен ли снимок памяти в ходе нормального функционирования системы или в момент ее критического сбоя, и именно поэтому отладка в режиме *post mortem* вполне подходит для выполнения подобных исследований. Однако некоторые чрезвычайно интересные сведения можно получить также и в процессе изучения содержимого системной памяти нормально функционирующей «живой» системы. Для слежения за содержимым системной памяти прямо в процессе ее функционирования мы будем использовать специальный драйвер режима ядра, но подробнее об этом будет рассказано в следующих главах книги.

Итак, в момент возникновения критического сбоя по вашему приказу система копирует полное содержимое памяти в специальный файл, который называют английским термином *dump file* или *crash dump*. Можно сделать вывод, что размер этого файла будет совпадать с объемом физической памяти, установленной на компьютере. Если говорить точнее, то размер этого файла не намного меньше, чем объем физической памяти компьютера. Запись содержимого оперативной памяти на диск выполняется специальной подпрограммой ядра в процессе обработки возникшего исключения. Однако обработчик исключения не осуществляет запись содержимого памяти напрямую в файл. Вместо этого содержимое физической памяти прежде всего перемещается в файл виртуальной памяти, который является частью системного диспетчера управления памятью. В дальнейшем, в процессе следующей начальной загрузки, система пытается скопировать снимок физической памяти в момент сбоя из файла виртуальной памяти в отдельный файл. Почему создание файла, содержащего снимок физической памяти в момент сбоя, требуется осуществлять в два этапа? Считается, что в момент сбоя встроенные в ОС механизмы работы с файлами могут оказаться недееспособными, поэтому в момент сбоя для сохранения содержимого памяти на диске используется диспетчер виртуальной памяти. Однако при этом вы должны позаботиться о том, чтобы размер файла виртуальной памяти был как минимум в два раза больше объема установленной на компьютере физической памяти. Но почему в два раза? Неужели файла, размер которого равен объему физической памяти, будет недостаточно? Конечно, нет, ведь в файле такого размера можно сохранить только снимок физической памяти в момент сбоя и ничего больше. Не стоит забывать о том, что копирование этого снимка из файла виртуальной памяти в отдельный файл выполняется в процессе начальной загрузки системы. Именно на этом этапе система нуждается в дополнительной виртуальной памяти, и если размер файла виртуальной памяти будет недостаточным, на экране будут появляться сообщения low on virtual memory (мало

виртуальной памяти). Чтобы избежать подобных проблем, необходимо заранее должным образом настроить размер файла виртуальной памяти таким образом, чтобы он в два раза превышал объем установленной на компьютере физической памяти.

Принимая во внимание все вышесказанное, вы должны открыть панель управления Windows 2000 и изменить значения следующих параметров:

- Увеличьте суммарный объем файлов виртуальной памяти в вашей системе таким образом, чтобы он был как минимум в два раза больше объема физической памяти, установленной на компьютере. Для этой цели откройте раздел System (Система) панели управления, перейдите на вкладку Advanced (Дополнительно) и щелкните на кнопке Performance Options (Параметры быстродействия). В разделе Virtual memory (Виртуальная память) щелкните на кнопке Change (Изменить). После этого измените число в графе Maximum size (Максимальный размер) таким образом, чтобы оно соответствовало приемлемой для вас конфигурации. На рис. 1.1 показана конфигурация виртуальной памяти

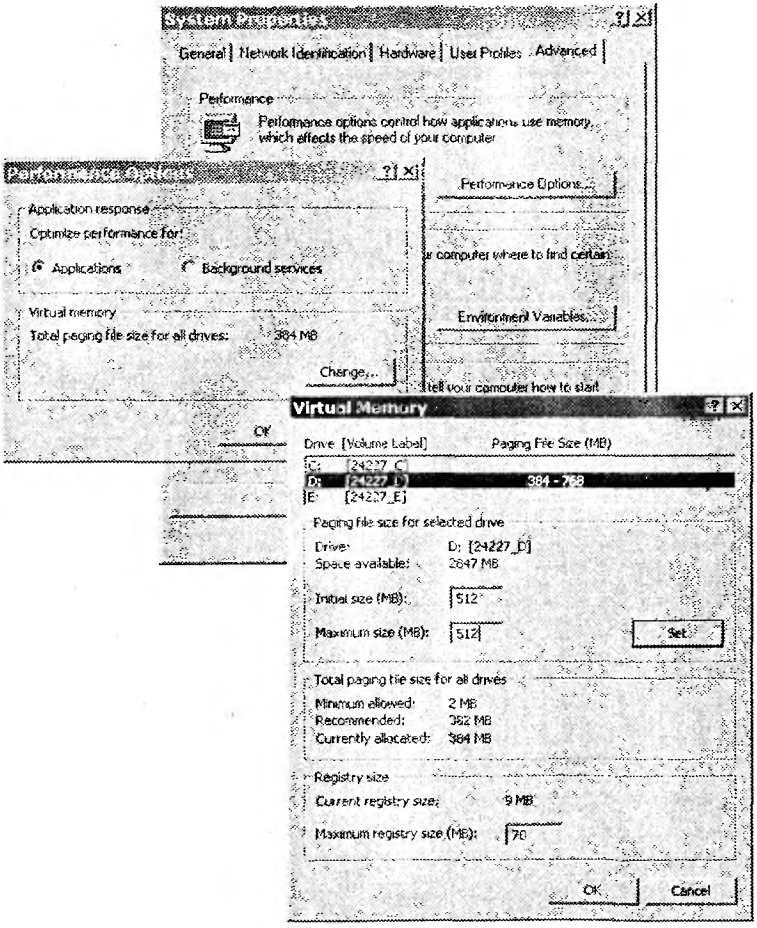

Рис. 1.1. Настройка размера файла виртуальной памяти

компьютера, на котором я пишу эти строки. Внутри моего компьютера установлено 256 Мбайт, поэтому размер файла виртуальной памяти должен быть не меньше 512 Мбайт. Изменив значение параметра, щелкните на кнопке Set (Установить) и при помощи кнопки OK закройте все открытые окна, за исключением окна System Properties (Свойства системы).

- После этого следует настроить систему таким образом, чтобы при возникновении критических ошибок снимок физической памяти записывался в файл. Для этого в окне диалога System Properties (Свойства системы) необходимо щелкнуть на кнопке Startup and Recovery (Загрузка и восстановление) и перейти к разделу Write Debugging Information (Запись отладочной информации). В ниспадающем списке необходимо выбрать позицию Complete Memory Dump (Полный дамп памяти), а в графе Dump File (Файл дампа памяти) необходимо указать путь и имя файла, в который будет занесен снимок оперативной памяти системы в момент сбоя. По умолчанию снимок записывается в файл %SystemRoot%\MEMORY.DMP (рис. 1.2). При помощи флажка Overwrite any existing file (Затирать существующий файл) вы можете приказать системе перезаписывать существующий файл. Выполнив настройку, необходимо закрыть все открытые окна при помощи кнопки OK.

Как нарушить работу системы?

Теперь, когда система настроена на сохранение снимка оперативной памяти в файле, необходимо выполнить самое страшное, с чем может столкнуться системный программист Windows 2000: давайте организуем критический сбой системы. Многие попадали в ситуацию, когда голубой экран смерти (BSOD) раз за разом появляется перед уставшими глазами, в то время как до крайнего срока сдачи работы остаются считанные часы. Но сейчас, когда вы намерены умышленно организовать критический сбой, наверняка под рукой не найдется ни одной программы, способной «убить» систему. Можно попробовать прием, описанный Дэвидом Соломоном (David Solomon) во втором издании «Inside Windows NT». В этой книге Дэвид пишет: «Существует ли беспроигрышный метод создания снимка оперативной памяти в момент сбоя? Для этого достаточно, используя утилиту kill.exe из комплекта Windows NT Resource Kit, просто уничтожить процесс подсистемы Win32 (csrss.exe) или процесс подключения к Windows NT (winlogon.exe). (Для этого вы должны обладать полномочиями администратора системы.)». Попробуем воспользоваться советом Дэвида. Сюрприз! В Windows 2000 этот прием уже не действует. В общем и целом, это неплохая новость: новую ОС из семейства NT уже нельзя легко разрушить при помощи небольшой простой утилиты, официально распространяемой Microsoft. Слава богу, в Windows 2000 эта зияющая дыра в системе безопасности наконец-то закрыта. Но как же теперь можно нарушить работу системы для того, чтобы получить снимок ее оперативной памяти? Настало время вспомнить простое старое правило NT: если задача кажется,невыполнимой в среде Win32, достаточно написать драйвер режима ядра — такой драйвер справится

с любой, даже самой запретной задачей. Операционная система Windows 2000 управляет прикладными программами чрезвычайно осторожно: между ядром системы и прикладными программами возводится непреодолимая стена, и любое приложение, пытающееся преодолеть этот барьер, безжалостно уничтожается. Такое положение дел оправдано с точки зрения безопасности системы, но является проблемой для программиста, разрабатывающего код, который должен напрямую работать с оборудованием компьютера. В отличие от DOS, в которой любая программа может делать с любыми аппаратными устройствами все, что ей вздумается, Windows 2000 строго контролирует прямой доступ к оборудованию. Напрямую с аппаратными устройствами компьютера могут работать только программные модули специального вида, которые называются *драйверами режима ядра* (kernel-mode drivers).

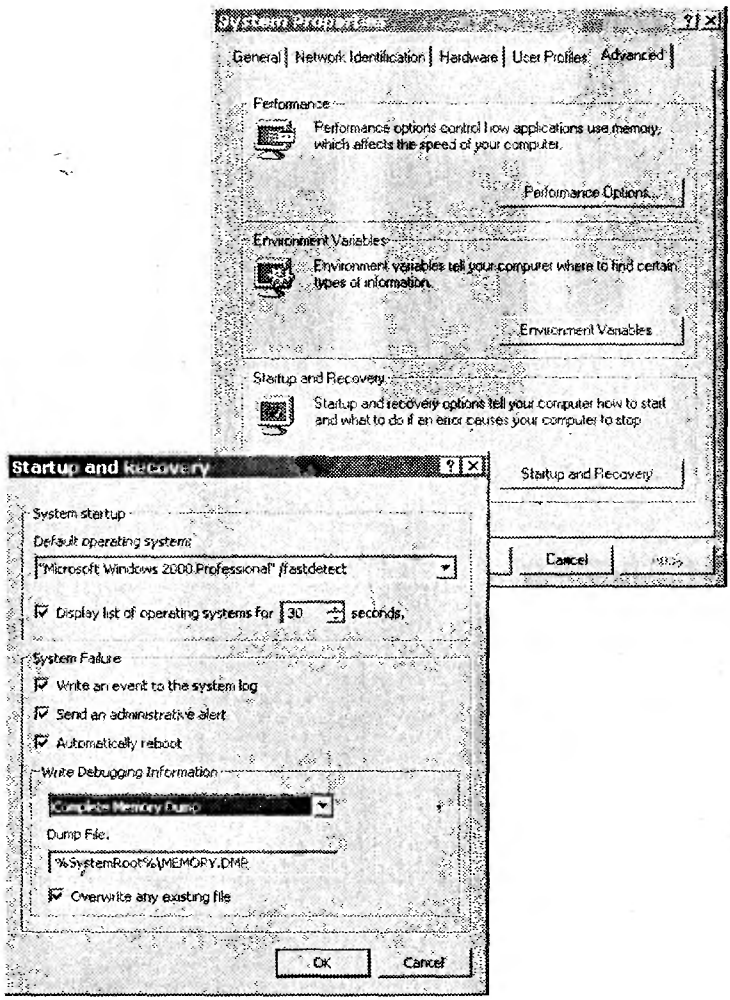

Рис. 1.2. Настройка параметров снимка оперативной памяти

В главе 3 я планирую познакомить читателей с основами разработки драйверов режима ядра. Однако сейчас не буду отвлекаться от основной темы обсуждения и скажу лишь, что разрушение системы — это самая несложная из задач, которые под силу драйверам режима ядра. В Windows 2000 отсутствует какой-либо механизм блокирования драйверов, работающих некорректно. Как только драйвер выполняет любую, пусть даже самую безобидную, но запрещенную инструкцию, система мгновенно останавливает работу и отображает голубой экран смерти. Самой простой и самой безопасной запрещенной операцией является чтение памяти по неправильному адресу. В частности, система явно перехватывает любые обращения к памяти, на которую указывает указатель NULL (обращение к ячейке по указателю NULL является одной из наиболее распространенных ошибок в программировании на C), поэтому для того, чтобы вызвать критический сбой системы, достаточно осуществить чтение ячейки, на которую указывает указатель NULL. Именно эту операцию выполняет драйвер w2k_kill.sys, записанный на прилагаемом к книге компакт-диске. Именно этот несложный драйвер будет одним из первых драйверов режима ядра, рассмотренных в данной книге.

В листинге 1.1 содержится фрагмент исходного кода этого драйвера. Функция DriverEntry(), тело которой определяется в файле w2k_kill.c, содержит всего один оператор, который осуществляет чтение по нулевому адресу. При выполнении этой операции система отображает голубой экран смерти (BSOD). Разрабатывая подобный бессмысленный код, следует принимать во внимание, что чрезвычайно мощный механизм оптимизации, встроенный в Visual C/C++, может свести на нет все ваши попытки вставить в исполняемый код программы бессмысленные инструкции. Оптимизатор тщательно анализирует исходный код и пытается исключить из результирующего исполняемого кода любые инструкции, в результате исполнения которых не возникает никакого осмысленного постоянного эффекта. При оптимизации рассматриваемой функции DriverEntry() у оптимизатора связаны руки, так как данная функция использует значение, извлеченное из ячейки по нулевому адресу, в качестве возвращаемого значения. Это означает, что содержимое ячейки с нулевым адресом следует переместить в процессорный регистр EAX. Для этого проще всего воспользоваться процессорной инструкцией MOV EAX, [0]. Именно эта инструкция вызывает исключение, останавливающее работу системы.

Чтобы загрузить и запустить драйвер w2k_kill.sys, можно воспользоваться утилитой загрузки/выгрузки драйверов под названием w2k_load.exe, исходный код которой рассматривается в главе 3. Если вы готовы нарушить функционирование вашей системы Windows 2000, выполните следующее:

1. Закройте все приложения.
2. Вставьте в привод CD-ROM прилагаемый к книге компакт-диск.
3. В меню Start (Пуск) выберите команду Run (Выполнить).
4. В графе ввода команды введите d:\bin\w2k_load w2k_kill.sys, при необходимости замените букву d: латинской буквой, обозначающей ваш привод CD-ROM. Щелкните кнопку OK.

В результате этих действий утилита w2k_load.exe попытается загрузить в память драйвер w2k_kill.sys, расположенный в каталоге \bin на компакт-диске. Как только управление будет передано функции DriverEntry() драйвера, система отобразит голу-

бой экран смерти (BSOD). На экране вашего монитора появится сообщение, подобное тому, что изображено на рис. 1.3. По мере того как содержимое памяти будет сбрасываться в файл виртуальной памяти на жесткий диск, вы увидите, как значение счетчика увеличивается от 0 до 100 (или около того). Если в окне диалога Startup and Recovery (Загрузка и восстановление) установлен флажок Automatically reboot (Выполнить автоматическую перезагрузку) (см. рис. 1.2), сразу же после переноса содержимого физической памяти в виртуальную произойдет перезапуск системы. Когда система выведет на экран приглашение на вход в систему, подождите некоторое время, пока перестанет мерцать индикатор обращения к жесткому диску. Мерцание индикатора означает, что система переносит снимок оперативной памяти из виртуальной памяти в отдельный файл, местоположение и имя которого указываются в окне диалога Startup and Recovery (Загрузка и восстановление). Если в это время вы тем или иным образом потревожите систему (например, не дождавшись завершения процесса переноса, завершите работу системы), полученный в результате файл снимка оперативной памяти окажется некорректным и отладчик ядра Kernel Debugger не сможет с ним работать.

Листинг 1.1. В результате чтения ячейки с нулевым адресом в режиме ядра происходит критический сбой системы

```
NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING puRegistryPath)
 {
 return *((NTSTATUS *) 0); // чтение указателя на NULL
 }
```

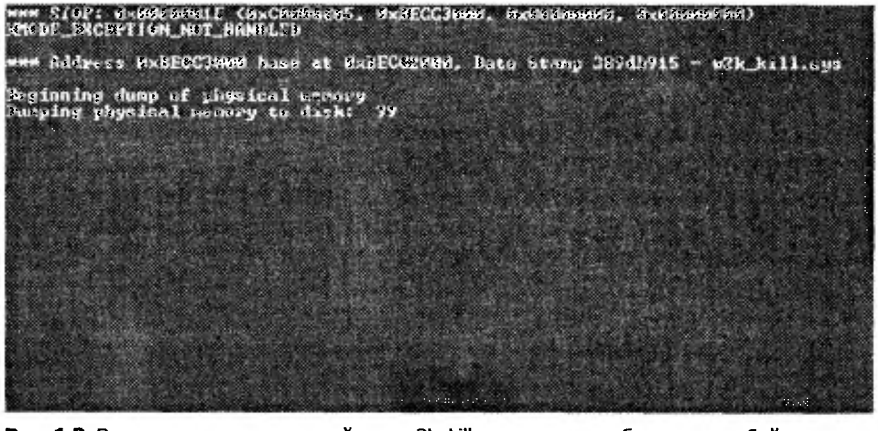

Рис. 1.3. В результате загрузки драйвера w2k_kill.sys система отображает голубой экран смерти (BSOD)

На рис. 1.3 показано имя модуля, вызвавшего исключение (w2k_kill.sys), а также адрес инструкции, ставшей причиной сбоя (0xBECC3000). В вашей системе этот адрес, скорее всего, будет другим, так как конкретное значение определяется аппаратной конфигурацией компьютера. Адрес, по которому в память загружается драйвер, как и адрес загрузки библиотеки DLL, не является чем-то жестко заданным и может варьироваться от загрузки к загрузке. Организовав критический сбой системы, вы должны записать этот адрес для того, чтобы воспользоваться им позже, при работе с отладчиком ядра Kernel Debugger.

Полагаю, нелишним будет небольшое предупреждение: не стоит слишком часто намеренно устраивать сбой системы. Но что может произойти страшного в результате загрузки столь безобидного драйвера, как w2k_kill.sys? Конечно, код этого драйвера сам по себе не может причинить системе никакого непоправимого вреда, однако неприятности могут возникнуть в случае, если драйвер будет загружен в неподходящий момент. Представьте себе, что чтение ячейки с адресом NULL происходит в момент, когда другой программный поток занят выполнением какой-то важной процедуры. В этом случае система прекратит нормальное функционирование еще до того, как поток успеет корректно завершить начатое дело. Например, если вы загрузили драйвер w2k_kill.sys в момент, когда механизм Active Desktop занимается чем-то важным, после перезагрузки вы увидите на экране пугающее сообщение, а также предупреждение о том, что система нуждается в восстановлении. Поэтому, прежде чем устраивать критический сбой системы, убедитесь в том, что не выполняется обработка каких-либо важных данных и что вся хранящаяся в кэше информация сохранена на диске.

ВНИМАНИЕ

Имейте в виду, что ни автор, ни издатель данной книги не несут никакой ответственности за какой-либо ущерб, нанесенный вам в результате загрузки драйвера w2k_kill.sys.

Установка файлов символьных идентификаторов

Перезагрузив систему, вы обнаружите в указанном вами месте на локальном диске файл, в котором содержится снимок оперативной памяти Windows 2000, в котором среди прочего исполняемого кода можно найти также и код сбойного драйвера в момент выполнения чтения ячейки с адресом NULL. Содержимое этого файла в точности совпадает с содержимым оперативной памяти в момент сбоя, поэтому его можно изучать так, как будто это память нормально функционирующей системы. Конечно же, снимок оперативной памяти, сохраненный в файле на жестком диске, нельзя назвать полноценным объектом для исследований. Скорее он напоминает труп животного — система находится в остановленном состоянии и не может реагировать на внешние раздражители. Однако это не должно вас беспокоить. Для того чтобы приступить к анализу снимка, необходимо установить в системе так называемые *файлы символьных идентификаторов* (symbol files). Эти файлы используются отладчиком ядра для удобства работы со снимком оперативной памяти.

Если вы являетесь подписчиком MSDN, вы можете установить файлы символьных идентификаторов, воспользовавшись компакт-диском *Windows 2000 Customer Support — Diagnostic Tools*, который входит в состав темно-зеленого набора компакт-дисков под общим названием *Development Platform (English)*. После того как вы вставите компакт-диск в привод CD-ROM, автоматически запустится Internet Explorer и вы увидите на экране содержимое HTML-странички с именем \DBG.HTM. При помощи этой странички вы можете выбрать один из нескольких вариантов установки. Если вы используете окончательную (free) версию Windows 2000, следует выбрать вариант Install retail symbols (установить окончательную версию символьных файлов). Если же вы работаете с проверочной (checked) версией Windows 2000, необходимо выбрать Install debug symbols (установить отладочную версию символь-

ных файлов). Установить файлы символьных идентификаторов можно и без обращения к страничке \DBG.HTM, для этого достаточно запустить одну из программ установки: \SYMBOLS\I386\RETAIL\SYMBOLSX.EXE (рис. 1.4) или \SYMBOLS\I386\DEBUG\SYMBOLSX.EXE. Именно на эти исполняемые файлы указывают гиперссылки, расположенные на страничке \DBG.HTM. Утилита установки скопирует несколько DBG- и PDB-файлов из архива SYMBOLS.CAB в различные подкаталоги корневого системного каталога. По умолчанию символьные файлы копируются в подкаталог с именем %systemroot%\Symbols. Токен %systemroot% обозначает имя каталога, в котором установлена операционная система Windows 2000. В рассматриваемом далее примере этот каталог имеет имя D:\WINNT.

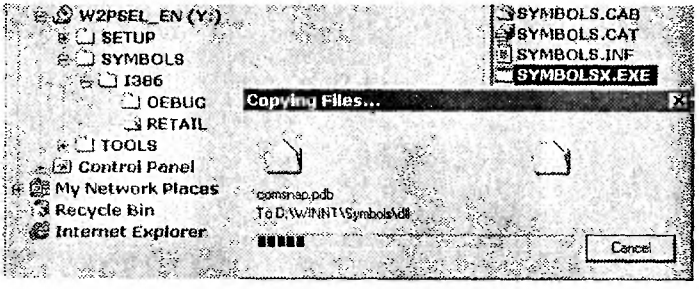

Рис. 1.4. Установка отладочных версий файлов символьных идентификаторов

При запуске отладчик Windows 2000 Kernel Debugger пытается обнаружить файлы символьных идентификаторов при помощи переменной окружения _NT_SYMBOL_PATH (обратите внимание на символ подчеркивания, стоящий в начале имени переменной). Чтобы облегчить ему эту задачу, вы можете прямо сейчас определить эту переменную. Откройте раздел System (Система) панели управления и перейдите на вкладку Advanced (Дополнительно), после чего щелкните на кнопке Environment Variables (Переменные среды). В открывшемся окне щелкните на кнопке New (Создать) в разделе System variables (Системные переменные), а затем введите имя и значение новой переменной в графах Variable Name (Имя переменной) и Variable Value (Значение переменной). Эта процедура проиллюстрирована на рис. 1.5. Имейте в виду, что D:\WINNT необходимо заменить на путь, хранящийся в переменной %systemroot%. После того как вы закроете все открытые окна при помощи кнопок OK, вы можете считать установку файлов символьных идентификаторов завершенной.

Какое значение должно быть присвоено переменной окружения _NT_SYMBOL_PATH? Документация Microsoft дает противоречивые ответы на этот вопрос. В разделах документации DDK, посвященных отладке ядра, говорится, что в состав значения _NT_SYMBOL_PATH должно входить имя подкаталога symbols. Таким образом, значение этой переменной должно быть d:\WINNT\SYMBOLS (или эквивалент, уместный для вашей конфигурации системы). Однако в документации пакета Platform Software Development Kit (SDK), где речь идет о библиотеке dbghlp.dll, определение пути к файлам символьных идентификаторов описывается несколько иначе:

«Для того чтобы обнаружить символьные идентификаторы, применяемые в процессе отладки (файлы с расширением DBG), отладочная библиотека использует путь поиска файлов символьных идентификаторов, при этом к пути поиска до-

бавляется строка \symbols, а затем строки \dll, \exe и \sys для файлов DLL, EXE и SYS соответственно. Например, типичным местоположением файлов символьных идентификаторов для библиотек DLL является каталог c:\mysymbols\symbols\dll, а файлы символьных идентификаторов для EXE-файлов, как правило, располагаются в каталоге c:\mysymbols\symbols\dll. [...]

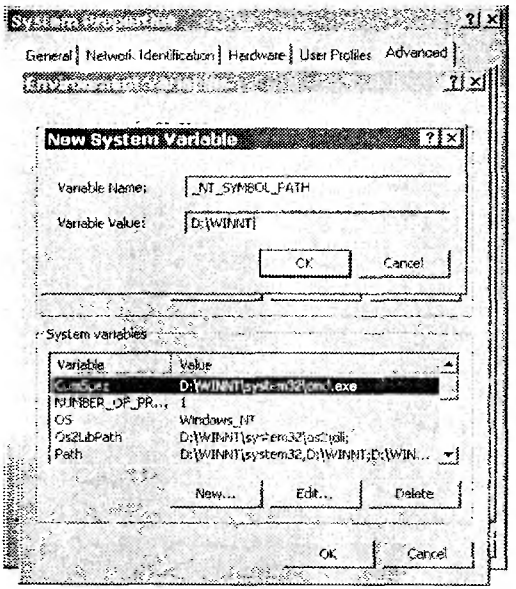

Рис. 1.5. Определение переменной окружения _NT_SYMBOL_PATH

Если вы определили переменные окружения _NT_SYMBOL_PATH или _NT_ALT_SYMBOL_PATH, поиск файлов символьных идентификаторов будет осуществляться в следующем порядке:

1. Текущий рабочий каталог приложения.
2. Путь, указанный в переменной окружения _NT_SYMBOL_PATH.
3. Путь, указанный в переменной окружения _NT_ALT_SYMBOL_PATH.
4. Путь, указанный в переменной окружения SYSTEMROOT».

(Библиотека MSDN. Апрель 2000 \ Platform SDK \ Base Services \ Debugging and Error Handling \ Debug Help Library \ About DbgHelp \ Symbol Handling \ Symbol Paths.)

Можно сделать вывод, что значение переменной _NT_SYMBOL_PATH должно быть равно скорее d:\winnt, а не d:\winnt\symbols. Чтобы понять, какое из утверждений правильно, я попробовал оба варианта и с удовольствием отметил, что оба они работают — отладчик Kernel Debugger обнаруживает файлы символьных идентификаторов вне зависимости от того, какой из вариантов настройки переменной _NT_SYMBOL_PATH вы предпочтете. Однако не следует думать, что значение переменной _NT_SYMBOL_PATH может быть абсолютно любым: попробуйте внести в эту переменную неправильный путь, и отладчик не сможет обнаружить файлы символьных идентификаторов.

Подготовка к работе отладчика Kernel Debugger

Теперь для того, чтобы приступить к анализу снимка оперативной памяти, вам осталось только лишь установить в системе и настроить отладчик Kernel Debugger. Если в системе установлен пакет Windows 2000 DDK, вы можете обнаружить отладчики в подкаталоге \NTDDK\bin. Исполняемый файл отладчика Kernel Debugger имеет имя i386kb.exe. Если же пакет Windows 2000 DDK не установлен, вы можете установить средства отладки с компакт-диска *Windows 2000 Customer Support — Diagnostic Tools*, входящего в состав библиотеки MSDN. Именно с этого компакт-диска вы устанавливали файлы символьных идентификаторов. Чтобы установить средства отладки с компакт-диска MSDN, достаточно сделать щелчок на гиперссылке Install Debugging Tools (установить средства отладки) странички \DBG.HTM или напрямую запустить исполняемый файл \TOOLS\I386\DBGPLUS.EXE, на который указывает эта ссылка. Программа установки скопирует средства отладки в каталог с именем \Program Files\Debuggers\bin.

После установки отладчика Kernel Debugger рекомендуется сделать ярлык, при помощи которого исполняемый файл i386kb.exe запускается с нужными вам параметрами командной строки. В частности, если вы планируете использовать отладчик для анализа снимка оперативной памяти, полученного в результате запуска драйвера w2k_kill.sys, при запуске отладчика следует применить ключ командной строки -z, после которого следует указать полное имя файла, в котором хранится интересующий вас снимок. Получив такую команду, отладчик Kernel Debugger загрузит указанный вами файл в оперативную память. Процедура создания ярлыка, указывающего на отладчик Kernel Debugger, показана на рис. 1.6.

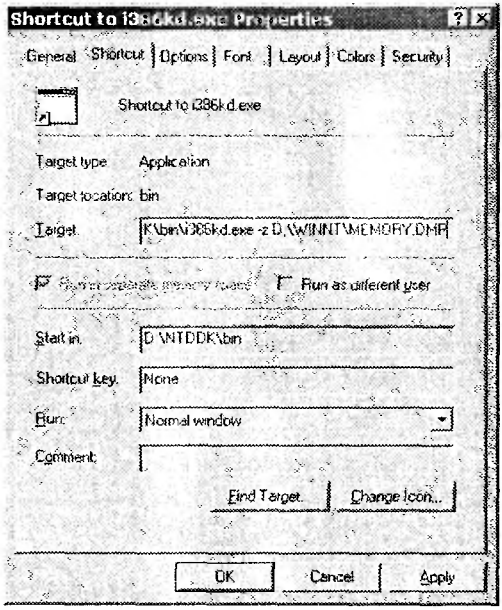

Рис. 1.6. Создание ярлыка, указывающего на отладчик Kernel Debugger

Наконец-то все готово к первому сеансу отладки. Сделав двойной щелчок на ярлыке, указывающем на исполняемый файл отладчика, вы увидите перед собой консольное окно, подобное изображенному на рис. 1.7. Приглашение kb> и мигающий курсор указывают на то, что отладчик Kernel Debugger готов к исполнению ваших команд. Прежде чем приступать к каким-либо действиям, убедитесь в том, что путь поиска файлов символьных идентификаторов, отображаемый ниже сообщения о правах на копирование, указывает на корректный каталог. Если этот путь указан неправильно, скорее всего, при настройке соответствующей переменной окружения вы допустили ошибку (см. рис. 1.5). Сообщение, отображенное на экране при запуске отладчика, указывает также на то, что в процессе запуска были загружены три библиотеки DLL расширения. Отладчик i386kb.exe обладает мощным механизмом расширения своих возможностей. Благодаря использованию этого механизма вы можете расширить стандартный набор команд этого отладчика дополнительными командами, реализованными при помощи дополнительных загружаемых DLL. Обозначения дополнительных команд отладчика начинаются с символа восклицательного знака (!), поэтому в англоязычной документации эти команды часто обозначают термином *bang commands* (восклицательные команды). Как вы увидите в дальнейшем, многие из этих команд чрезвычайно полезны.

На рис. 1.7 показан результат выполнения одной из встроенных команд: u becc3000. Символ u обозначает *unassemble* (дизассемблировать), а шестнадцатеричное число becc3000 — это стартовый адрес, с которого требуется выполнить дизассемблирование. По умолчанию отладчик рассматривает все числа как шестнадцатеричные, однако вы можете изменить основание системы счисления при помощи команды n. Например, если вы предпочитаете десятичную систему счисления, вы можете воспользоваться командой n 10. При этом вы также сможете пользоваться шестнадцатеричными числами, для этого перед числом необходимо разместить префикс 0x, позаимствованный из языка C. Адрес becc3000 — это именно тот адрес, по которому произошел сбой драйвера w2k_kill.sys (см. рис. 1.3). Как уже отмечалось ранее, в вашей системе этот адрес, скорее всего, будет другим. Воспользуйтесь командой u для того, чтобы дизассемблировать участок памяти по адресу, который ваша система отобразила на голубом экране смерти (BSOD) в качестве адреса сбойной команды. Несмотря на то что в вашей системе этот адрес, скорее всего, будет другим, как и у меня, у вас по этому адресу должна располагаться машинная инструкция mov eax,[00000000] (рис. 1.7). Если это не так, значит, скорее всего, вы изучаете не тот снимок оперативной памяти. В этом случае вы должны еще раз проверить ярлык, который вы используете для запуска отладчика (см. рис. 1.6). Машинная инструкция mov eax,[00000000] перемещает значение из ячейки ОЗУ с адресом 0x00000000 в процессорный регистр EAX. Именно эта команда реализует выражение return *((NTSTATUS *) 0) языка C, представленное в листинге 1.1 и обозначающее операцию чтения ячейки с адресом NULL. Для ошибок данного типа не существует какого-либо специального обработчика, поэтому, как показано на рис. 1.3, на голубом экране смерти система отображает ошибку типа KMODE_EXCEPTION_NOT_HANDLED. При желании вы можете подробнее узнать об этом типе ошибок в статье «More On Kernel Debugging: KMODE_EXCEPTION_NOT_HANDLED», опубликованной в шестом (ноябрь/декабрь) номере журнала «The NT Insider» за 1999 год.

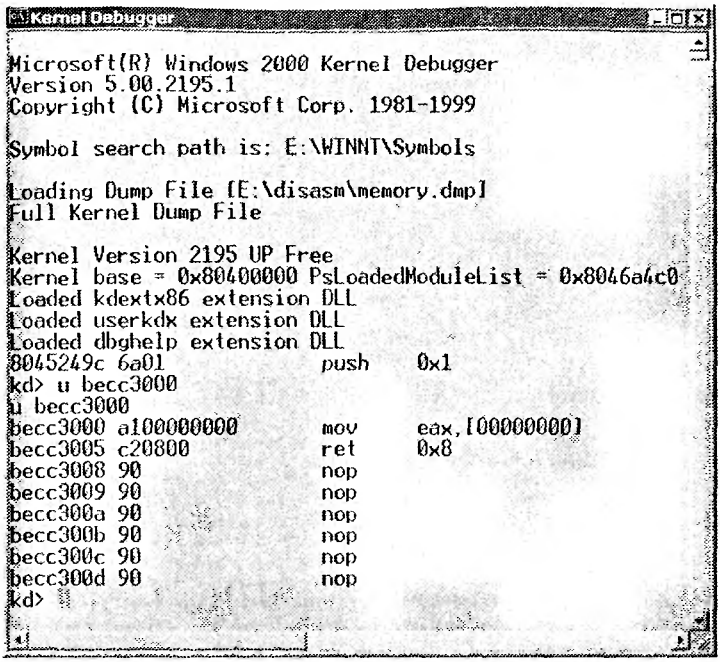

Рис. 1.7. Начало отладочного сеанса Kernel Debugger

Команды отладчика Kernel Debugger

Для удобства работы с отладчиком большинство его команд обозначается одним, двумя или тремя символами, однако иногда попадаешь в ситуацию, когда сложно вспомнить ту или иную команду. Для удобства читателей все стандартные команды отладчика собраны в приложении А. Таблица А.1 является кратким справочником по командам отладчика Kernel Debugger. Эта таблица представляет собой отредактированную версию электронной справки, которую отладчик выводит на экран по команде ?. Типы аргументов, используемых совместно с командами отладчика, описываются в табл. А.2.

Как уже упоминалось, отладчик ядра Kernel Debugger позволяет выполнять внешние команды, реализованные при помощи внешних подключаемых к отладчику DLL-файлов. Обозначения этих команд начинаются с символа восклицательного знака. Когда программист отдает подобную команду, отладчик ищет имя команды в списках экспорта загруженных DLL расширения. На рис. 1.7 видно, что в начале работы отладчик Kernel Debugger загрузил три DLL расширения: kdextx86.dll, userkdx.dll и dbghelp.dll в указанном порядке. Последняя из них располагается в том же каталоге, что и исполняемый файл i386kb.exe. Две другие библиотеки расширения доступны в четырех версиях: окончательная и отладочная для Windows NT 4.0 (подкаталоги nt4fre и nt4chk), а также окончательная и отладочная для Windows 2000 (подкаталоги w2kfre и w2kchk). Как правило, пытаясь обнаружить подпрограмму выполнения той или иной дополнительной команды, отладчик просматривает все DLL

расширения по порядку, пока не найдет в списке экспорта очередной DLL введенную программистом команду. Однако при желании вы можете явно указать модуль, к которому принадлежит введенная вами дополнительная команда. Имя модуля вводится между восклицательным знаком и обозначением команды и отделяется от обозначения команды символом точки. Например, команда help обрабатывается как в модуле kdextx86.dll, так и в модуле userkdx.dll. Если вы наберете на клавиатуре просто !help, на экране появится электронная справка модуля kdextx86.dll. Чтобы приказать отладчику выполнить команду help модуля userkdx.dll, необходимо набрать !userkdx.help (или !userkdx.help -v в случае, если вы хотите получить более подробную справочную информацию). Кстати говоря, если вы изучите необходимые для этого правила, вы сможете самостоятельно написать свое собственное расширение отладчика. Подробные инструкции о том, как это сделать, содержатся в статье «Make WinDbg Your Friend: Creating Debugger Extentions», опубликованной в 5-м (сентябрь/октябрь) номере журнала «The NT Insider» за 1999 год. В первую очередь в статье речь идет об отладчике WinDbg.exe, однако оба отладчика (как WinDbg.exe, так и i386kb.exe) используют одни и те же программные интерфейсы и совместимы с одними и теми же DLL расширения, поэтому большая часть изложенной в статье информации применима также и в отношении i386kb.exe.

Электронная справка модулей kbextx86.dll и userkbx.dll приведена в табл. А.3 и А.4 соответственно. Можно заметить, что в этих таблицах содержится значительно большее количество команд, чем упоминается в документации DDK, кроме того, многие документированные команды обладают некоторыми дополнительными недокументированными в DDK параметрами.

Десять наиболее часто используемых команд отладчика

Таблицы от А.1 до А.4 демонстрируют, что количество команд отладчика Kernel Debugger, а также его стандартных расширений, воистину огромно. В данном разделе я подробно рассмотрю некоторые из команд, которые могут быть особенно полезными при исследовании внутренностей Windows 2000.

u: Дизассемблировать машинный код

Вы уже использовали эту команду для того, чтобы убедиться в том, что отладчик работает с правильным снимком оперативной памяти. Команда u имеет три формы:

1. u <начало> дизассемблирует восемь машинных инструкций начиная с адреса <начало>.
2. u <начало> <конец> дизассемблирует машинные инструкции начиная с адреса <начало> и заканчивая адресом <конец>. Машинная инструкция, расположенная по адресу <конец>, не включается в листинг.
3. u (без аргументов) начинает дизассемблирование с адреса, на котором закончено выполнение предыдущей команды u (вне зависимости от того, была ли предыдущая команда u с аргументами или без).

Конечно же, дизассемблирование больших участков кода при помощи этой команды может оказаться утомительным, однако команда u чрезвычайно удобна в случае, когда требуется быстро выяснить, что именно происходит по тому или иному адресу в памяти. Наверное, наиболее любопытной особенностью этой команды является возможность отображения имен символьных идентификаторов, на которые ссылается исполняемый код. Символьные идентификаторы отображаются даже в случае, если исследуемый вами модуль их не экспортирует. Рассматривая команду u отладчика Kernel Debugger, хочу отметить, что дизассемблирование исполняемых файлов Windows 2000 зачастую удобнее выполнять с использованием утилиты Multi-Format Visual Disassembler, о которой будет рассказано несколько позже. Утилиту Multi-Format Visual Disassembler можно обнаружить на компакт-диске, прилагаемом к данной книге.

db, dw, dd: Шестнадцатеричный листинг содержимого памяти в виде значений BYTE, WORD или DWORD

Если вас интересует участок памяти, в котором содержится не исполняемый код, а бинарные данные, для их изучения окажутся полезными команды просмотра шестнадцатеричных листингов. Для просмотра численных значений вы можете воспользоваться одной из команд db, dw или dd в зависимости от того, данные какого типа (BYTE, WORD или DWORD соответственно) содержатся по интересующему вас адресу.

- Команда db отображает содержимое памяти в двух панелях. На левой панели содержимое памяти отображается в виде двухзначных 8-битных шестнадцатеричных чисел. На правой панели содержимое памяти отображается в виде символов ASCII.
- Команда dw отображает содержимое памяти в виде четырехзначных 16-битных шестнадцатеричных чисел. Панель ASCII-символов не отображается.
- Команда dd отображает содержимое памяти в виде восьмизначных 32-битных шестнадцатеричных чисел. Панель ASCII-символов не отображается.

Совместно с каждой из этих команд можно использовать такой же набор аргументов, как и с командой u. Отличие состоит в том, что данные по адресу <конец> всегда включаются в список. Если аргументы не указаны, отображается следующий 128-байтовый блок.

x: Анализ символьных идентификаторов

Это чрезвычайно важная команда. Она выводит на экран список символьных идентификаторов, сформированный на основе содержимого файлов символьных идентификаторов. Команда x, как правило, используется в одной из трех форм:

- x *!* отображает список всех модулей, для которых можно просмотреть перечень символьных идентификаторов. Сразу же после запуска по умолчанию доступны только символьные идентификаторы модуля ntoskrnl.exe. Добавить новые модули или символьные идентификаторы можно при помощи команды .reload.

- x <модуль>!<фильтр> отображает список символьных идентификаторов, информация о которых содержится в файле символьных идентификаторов указанного модуля. Перечень идентификаторов фильтруется при помощи выражения <фильтр>, являющегося маской, в состав которой могут входить специальные символы ? и *. Имя модуля должно быть одним из имен, выводимых на экран по команде x *!*. Например, по команде x nt!* на экран будет выведена информация о символьных идентификаторах файла ntoskrnl.dbg, а команда x win32k!* выводит на экран перечень символьных идентификаторов файла win32k.dbg. Если в результате выполнения команды x ... отладчик выдает сообщение об ошибке Couldn't resolve 'x ...' (не могу выполнить 'x ...'), перезагрузите файлы символьных идентификаторов при помощи команды .reload и попытайтесь отдать команду x ... заново.
- x <фильтр> отображает все символьные идентификаторы, соответствующие маске <фильтр>. Эта форма команды x является разновидностью формы x <модуль>!<фильтр> только в ней опущена часть <модуль>!.

Помимо имен символьных идентификаторов команда x выводит на экран соответствующие виртуальные адреса. Для имени функции адрес соответствует точке входа в функцию, а для имени переменной адрес является базовым адресом этой переменной. Следует отметить, что команда x выводит на экран не только экспортируемые, но и внутренние, то есть закрытые для внешнего доступа, символьные идентификаторы модулей.

ln: Показать ближайшие символьные идентификаторы

Команда ln является моей любимой командой отладчика. Эта команда обеспечивает быстрый и удобный способ доступа к той информации о символьных идентификаторах, в которой вы больше всего нуждаетесь, изучая тот или иной участок памяти. Команда ln является идеальным дополнением к команде x. В то время как команда x позволяет вам получить листинг разнообразных символьных идентификаторов операционной системы, команда ln используется для поиска конкретного символьного идентификатора по адресу или имени.

- ln <адрес> отображает имя символьного идентификатора, который соответствует или предшествует указанному в команде адресу, а также имя идентификатора, следующего за указанным адресом.
- ln <имя> определяет адрес, соответствующий символьному идентификатору с указанным в команде именем, а далее действует так же, как команда ln <адрес>.

Как и в случае с командой x, команда ln позволяет работать как с экспортируемыми, так и с неэкспортируемыми символьными идентификаторами, поэтому она особенно полезна в ситуации, когда вы пытаетесь определить, в какое место памяти указывает тот или иной неизвестный указатель, обнаруженный вами в дизассемблерном коде или шестнадцатеричном листинге. Обратите внимание на то, что команды u, db, dw и dd позволяют вместо конкретных виртуальных адресов указывать имена символьных идентификаторов.

!processfields: Показать список полей структуры EPROCESS

Имя этой команды начинается с символа восклицательного знака, из чего можно заключить, что команда обрабатывается модулем расширения отладчика. И действительно, код обработки команды !processfields содержится в модуле kdextx86.dll. Команда отображает на экране имена и смещения полей формально не документированной структуры EPROCESS. Структура типа EPROCESS является объектом процесса и используется ядром для хранения сведений об исполняемом в системе процессе. Вывод команды !processfields показан в примере 1.1.

Пример 1.1. Строение структуры EPROCESS

```
kb> !processfields
!processfields
 EPROCESS structure offsets:
  Pcb:                                0x0
  ExitStatus:                         0x6c
  LockEvent:                          0x70
  LockCount:                          0x80
  CreateTime:                         0x88
  ExitTime                            0x90
  LockOwner:                          0x98
  UniqueProcessId:                    0x9c
  ActiveProcessLinks:                 0xa0
  QuotaPeakPoolUsage[0]:              0xa8
  QuotaPoolUsage[0]:                  0xb0
  PagefileUsage:                      0xb8
  CommitCharge:                       0xbc
  PeakPagefileUsage:                  0xc0
  PeakVirtualSize:                    0xc4
  VirtualSize:                        0xc8
  Vm:                                 0xd0
  DebugPort:                          0x120
  ExceptionPort:                      0x124
  ObjectTable:                        0x128
  Token:                              0x12c
  WorkingSetLock:                     0x130
  WorkingSetPage:                     0x150
  ProcessOutswapEnabled:              0x154
  ProcessOutswapped:                  0x155
  AddressSpaceInitialized:            0x156
  AddressSpaceDeleted:                0x157
  AddressCreationLock:                0x158
  ForkInProcess:                      0x17c
  VmOperation:                        0x180
  VmOperationEvent:                   0x184
  PageDyrectoryPte:                   0x1f0
  LastFaultCount:                     0x18c
  VadRoot:                            0x194
  VadHint:                            0x198
  CloneRoot:                          0x19c
  NumberOfPrivatePages:               0x1a0
  NumberOfLockedPages:                0x1a4
  ForkWasSuccessfull:                 0x182
  ExitProcessCalled:                  0x1aa
```

продолжение

Пример 1.1 (*продолжение*)

```
  CreateProcessReported:                  0x1ab
  SectionHandle:                          0x1ac
  Peb:                                    0x1b0
  SectionBaseAddress:                     0x1b4
  QuotaBlock:                             0x1b8
  LastThreadExitStatus:                   0x1bc
  WorkingSetWatch:                        0x1c0
  InheritedFromUniqueProcessId:           0x1c8
  GrantedAccess:                          0x1cc
  DefaultHardErrorProcessing              0x1d0
  LdtInformation:                         0x1d4
  VadFreeHint:                            0x1d8
  VdmObjects:                             0x1dc
  DeviceMap:                              0x1e0
  ImageFileName[0]:                       0x1fc
  VmTrimFaultValue:                       0x20c
  Win32Process:                           0x214
  Win32WindowStation:                     0x1c4
```

К сожалению, команда отображает только имена и смещения полей структуры, не показывая никакой информации о типах этих полей, однако, проанализировав взаимное расположение полей в структуре, вы можете сделать предположения о том, данные какого типа хранятся в том или ином поле. В частности, поле LockEvent располагается со смещением 0x70 относительно начала структуры, а смещение следующего поля составляет 0x80. Таким образом, размер поля LockEvent составляет 16 байт. Можно предположить, что в этом поле располагается структура типа KEVENT. Полагаю, что далеко не все читатели книги знают о том, что такое структура типа KEVENT, однако волноваться не стоит — я подробно расскажу о внутреннем строении объектов ядра в главе 7.

!threadfields: Показать список полей структуры ETHREAD

Это еще одна полезная команда, обрабатываемая кодом модуля kdextx86.dll. Команда !threadfields отображает имена и смещения еще одной формально не документированной структуры ядра ETHREAD. Структура ETHREAD используется ядром для представления программного потока. Вывод команды !threadfields показан в примере 1.2.

Пример 1.2. Строение структуры ETHREAD

```
kb> !threadfields
!threadfields
 ETHREAD structure offsets:
  Tcb:                                    0x0
  CreateTime:                             0x1b0
  ExitTime:                               0x1b8
  ExitStatus:                             0x1c0
  PostBlockList:                          0x1c4
  TerminationPostList:                    0x1cc
  ActiveTimerListLock:                    0x1d4
  ActiveTimerListHead:                    0x1d8
  Cid:                                    0x1e0
  LpcReplySemaphore:                      0x1e8
  LpcReplayMessage:                       0x1fc
  LpcReplayMessageId:                     0x200
  ImpersonationInfo:                      0x208
```

```
IrpList:                          0x20c
TopLevelIrp:                      0x214
ReadClusterSize                   0x21c
ForwardClusterOnly:               0x220
DisablePageFaultClustering:       0x221
DeadThread:                       0x222
HasTerminated:                    0x224
GrantedAccess:                    0x228
ThreadsProcess:                   0x22c
StartAddress:                     0x230
Win32StartAddress:                0x234
LpcExitThreadCalled:              0x238
HardErrorsAreDisabled:            0x239
```

!drivers: Отобразить перечень загруженных драйверов

Команда !drivers, обработку которой осуществляет модуль расширения kdextx86.dll, отображает на экране подробную информацию обо всех функционирующих модулях ядра и файловой системы. Если вы работаете со снимком оперативной памяти в момент сбоя, по этой команде отладчик отображает информацию о драйверах, которые функционировали в системе в момент сбоя. В примере 1.3 показан результат выполнения команды !drivers на моем компьютере. Обратите внимание на предпоследнюю строку списка (перед строкой TOTAL:). Эта строка соответствует драйверу w2k_kill.sys, ставшему причиной сбоя системы. Драйвер w2k_kill.sys загружен по адресу 0xBECC2000, именно это число можно обнаружить на голубом экране смерти, отображаемом системой в момент сбоя в результате загрузки этого драйвера (см. рис. 1.3).

Пример 1.3. Вывод на экран информации о загруженных системных модулях

```
kb> !drivers
!drivers
Loaded System Driver Summary

Base      Code    Size      Data    Size      Driver Name   Creation Time
80400000  142dc0  (1291 kb) 4d680   (309 kb)  ntoskrnl.exe  Wed Dec 08 00:41:11 1999
80062000  13c40   (  79 kb) 34e0    (  13 kb)  hal.dll       Sun Oct 31 00:48:14 1999
f0810000  1760    (   5 kb) 1000    (   4 kb)  BOOTVID.DLL   Thu Nov 04 02:24:33 1999
f0400000  bdc0    (  47 kb) 22a0    (   8 kb)  pci.sys       Thu Oct 28 01:11:08 1999
f0410000  99c0    (  38 kb) 18e0    (   6 kb)  isapnp.sys    Sat Oct 02 22:00:35 1999
f09c8000  760     (   1 kb) 520     (   1 kb)  intelide.sys  Fri Oct 29 01:20:03 1999
f0680000  42e0    (  16 kb) e80     (   3 kb)  PCIIDEX.SYS   Thu Oct 28 01:02:19 1999
f0688000  64a0    (  25 kb) a20     (   2 kb)  MountMgr.sys  Sat Oct 23 00:48:06 1999
bffe3000  192c0   ( 100 kb) 2b00    (  10 kb)  ftdisk.sys    Mon Nov 22 20:36:23 1999
f0900000  12e0    (   4 kb) 640     (   1 kb)  Diskperf.sys  Fri Oct 01 02:30:40 1999
[...]
bf255000  fc40    (  63 kb) 2120    (   8 kb)  wdmaud.sys    Wed Oct 27 20:40:45 1999
f0670000  9520    (  37 kb) 1f40    (   7 kb)  sysaudio.sys  Mon Oct 25 21:28:14 1999
f094c000  d40     (   3 kb) 860     (   2 kb)  ParVdm.sys    Tue Sep 28 05:28:16 1999
f0958000  a00     (   2 kb) 480     (   1 kb)  PfModNT.sys   Thu Dec 16 05:14:08 1999
bf0dd000  35520   ( 213 kb) 59e0    (  22 kb)  rv.sys        Tue Nov 30 08:38:21 1999
bf191000  d820    (  54 kb) 1280    (   4 kb)  Cdfs.sys      Mon Oct 25 21:23:52 1999
bed9a000  11f20   (  71 kb) 2ac0    (  10 kb)  ipsec.sys     Tue Nov 30 08:08:54 1999
beaaf000  0       (   0 kb) 0       (   0 kb)  ATMFD.DLL     Header Paged Out
be9eb000  16f60   (  91 kb) ccc0    (  51 kb)  kmixer.sys    Wed Nov 10 07:52:30 1999
becc2000  200     (   0 kb) a00     (   2 kb)  w2k_kill.sys  Sun Feb 06 19:10:29 2000
TOTAL:    79c660  (7793 kb) 15c160  (1392 kb)  (      0 kb      0 kb)
```

!sel: Анализ значений селекторов

По команде !sel, выполняемой кодом модуля расширения kdextx86.dll, на экран выводятся параметры селекторов процессорного устройства управления памятью. Если эта команда отдается без аргументов, на экран выводится информация о 16 последовательных селекторах в порядке увеличения номеров. Чтобы получить информацию обо всех селекторах, необходимо последовательно отдавать команду !sel до тех пор, пока отладчик не выдаст сообщение Selector is invalid (пример 1.4). Работа с селекторами будет подробно рассмотрена в главе 4. Там же я познакомлю читателей с примером программы, демонстрирующей использование селекторов на практике.

Пример 1.4. Отображение параметров селекторов

```
kb> !sel
!sel
0000  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0008  Bas=00000000 Lim=000fffff Pages DPL=0  P Code  RE A
0010  Bas=00000000 Lim=000fffff Pages DPL=0  P Data  RW A
0018  Bas=00000000 Lim=000fffff Pages DPL=3  P Code  RE A
0020  Bas=00000000 Lim=000fffff Pages DPL=3  P Data  RW A
0028  Bas=80244000 Lim=000020ab Bytes DPL=0  P TSS32    B
0030  Bas=ffdff000 Lim=00000001 Pages DPL=0  P Data  RW A
0038  Bas=00000000 Lim=00000fff Bytes DPL=3  P Data  RW A
0040  Bas=00000400 Lim=0000ffff Bytes DPL=3  P Data  RW
0048  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0050  Bas=80470040 Lim=00000068 Bytes DPL=0  P TSS32    A
0058  Bas=804700a8 Lim=00000068 Bytes DPL=0  P TSS32    A
0060  Bas=00022ab0 Lim=0000ffff Bytes DPL=0  P Data  RW A
0068  Bas=000b8000 Lim=00003fff Bytes DPL=0  P Data  RW
0070  Bas=ffff7000 Lim=000003ff Bytes DPL=0  P Data  RW
0078  Bas=80400000 Lim=0000ffff Bytes DPL=0  P Code  RE
kb> !sel
!sel
0080  Bas=80400000 Lim=0000ffff Bytes DPL=0  P Data  RW
0088  Bas=00000000 Lim=00000000 Bytes DPL=0  P Data  RW
0090  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0098  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00a0  Bas=814985a8 Lim=00000068 Bytes DPL=0  P TSS32    A
00a8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00b0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00b8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00c0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00c8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00d0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00d8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00e0  Bas=f0430000 Lim=0000ffff Bytes DPL=0  P Code  RE A
00e8  Bas=00000000 Lim=0000ffff Bytes DPL=0  P Data  RW
00f0  Bas=8042dce8 Lim=000003b7 Bytes DPL=0  P Code  EO
00f8  Bas=00000000 Lim=0000ffff Bytes DPL=0  P Data  RW
```

Завершение работы с отладчиком

Чтобы завершить работу отладчика и удалить его из памяти, достаточно закрыть окно консоли, в котором он работает. Однако корректным методом завершения

работы отладчика считается использование команды q. Как можно предположить, *q* обозначает *quit*, то есть «выход».

Дополнительные средства отладки

На прилагаемом к книге компакт-диске содержатся еще две чрезвычайно полезные программы, предназначенные для отладки исполняемого кода в среде Windows 2000. Обе эти программы публикуются мною с разрешения их авторов, являющихся моими друзьями, контакт с которыми я поддерживаю через Интернет. Я очень благодарен им за то, что они разрешили мне разместить полнофункциональные версии разработанных ими великолепных программ на компакт-диске, прилагаемом к написанной мною книге. Утилита *PE and COFF File Viewer* (PEview), разработанная Вейном Дж. Рэдберном (Wayne J. Radburn), является свободно распространяемой версией, специально отредактированной для читателей данной книги. Утилита *Multi-Format Visual Disassembler* (MFVDasm), написанная Джином-Луисом Сигне (Jean-Louis Seigne), является демо-версией. Она обладает всеми функциями и возможностями полноценной версии, однако время ее использования ограниченно. В данном разделе содержится краткое описание обеих программ.

MFVDasm: Multi-Format Visual Disassembler

Утилита MFVDasm — это не просто генератор ассемблерных листингов. Скорее эту утилиту следует рассматривать как мощное средство просмотра ассемблерного кода с некоторыми дополнительными, весьма полезными возможностями. На рис. 1.8 изображено рабочее окно программы MFVDasm в процессе анализа кода функции IoDetachDevice(), входящей в состав диспетчера ввода/вывода операционной системы Windows 2000. К сожалению, на рисунке не видно, насколько удобной является цветовая разметка, отображаемая в рабочем окне MFVDasm. В частности, все метки функций, а также именованных переходов и вызовов отображаются красным цветом. Переходы и вызовы, передающие управление по анонимным адресам (то есть по адресам, которые не ассоциированы с какими-либо экспортируемыми символьными идентификаторами), отображаются синим цветом, а ссылки на символьные идентификаторы, динамически импортируемые из других модулей, отображаются фиолетовым цветом. Все достижимые в рамках текущего модуля метки кода подчеркиваются, и это означает, что вы можете перейти к отмеченному меткой коду, щелкнув мышкой на соответствующей метке. Кнопки Back и Forward служат для облегчения перемещения по коду и используются подобно аналогичным кнопкам браузера Internet Explorer. Используя эти кнопки, вы можете с легкостью анализировать различные ветви исполнения отлаживаемой программы.

Правая панель рабочего окна предназначена для выбора символьного идентификатора или целевого адреса, к которому вы хотите перейти. Содержимое этой панели можно сортировать при помощи кнопок, расположенных в верхней части панели. В нижней части панели располагаются вкладки, позволяющие выбирать информацию, отображаемую на панели. Просмотр шестнадцатеричного листинга может

оказаться полезным в случае, если вы дизассемблируете код, в котором содержатся фрагменты текстовых строк. Программа MFVDasm может работать с очень большими файлами (например, с ntoskrnl.exe), чего не могут другие популярные дизассемблеры. Конечно же, вы можете записать анализируемый ассемблерный код в текстовый файл. В главном меню программы, а также в разнообразных контекстных меню, появляющихся на экране при щелчке правой кнопкой мыши на различных панелях программы, можно обнаружить многочисленные, подчас весьма полезные команды и возможности. Более подробная информация о программе MFVDasm располагается на домашней страничке этой программы по адресу http://redirect.to/MFVDasm.

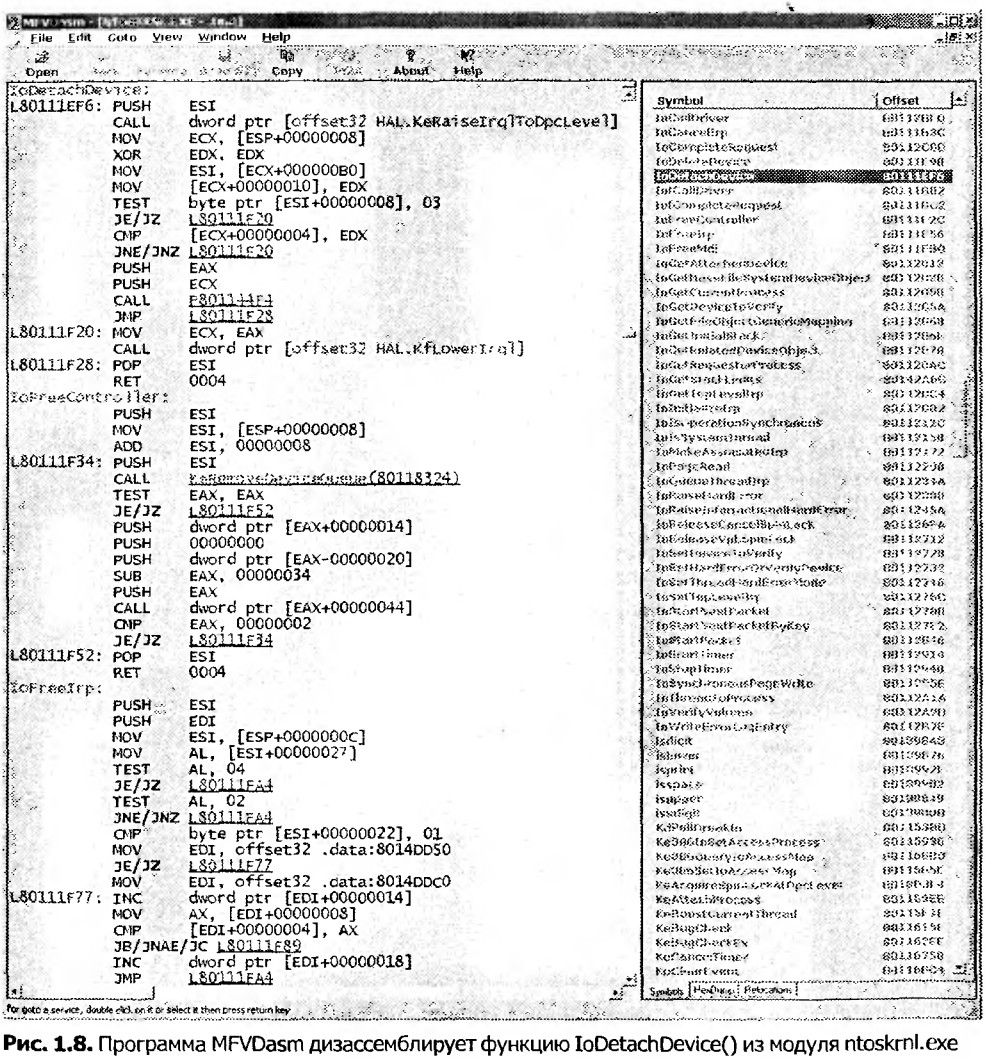

Рис. 1.8. Программа MFVDasm дизассемблирует функцию IoDetachDevice() из модуля ntoskrnl.exe

PEview: The PE and COFF File Viewer

Дизассемблер MFVDasm позволяет получить некоторые сведения о внутренней структуре файлов формата PE (Portable Executable), однако основное его предназначение — это дизассемблирование и просмотр исполняемого кода. Напротив, утилита PEview не позволяет просматривать дизассемблированный исполняемый код, однако с ее помощью вы можете досконально изучить внутреннюю структуру исполняемых файлов. Рабочее окно программы PEview, отображающее различные составные части модуля ntoskrnl.exe в виде дерева, показано на рис. 1.9. Узлы дерева выбираются на левой панели рабочего окна, в то время как на правой панели отображается связанная с выбранным узлом информация. На рис. 1.9 выбрана структура IMAGE_OPTIONAL_HEADER, входящая в состав структуры IMAGE_NT_HEADERS, расположенной в начале исполняемого файла.

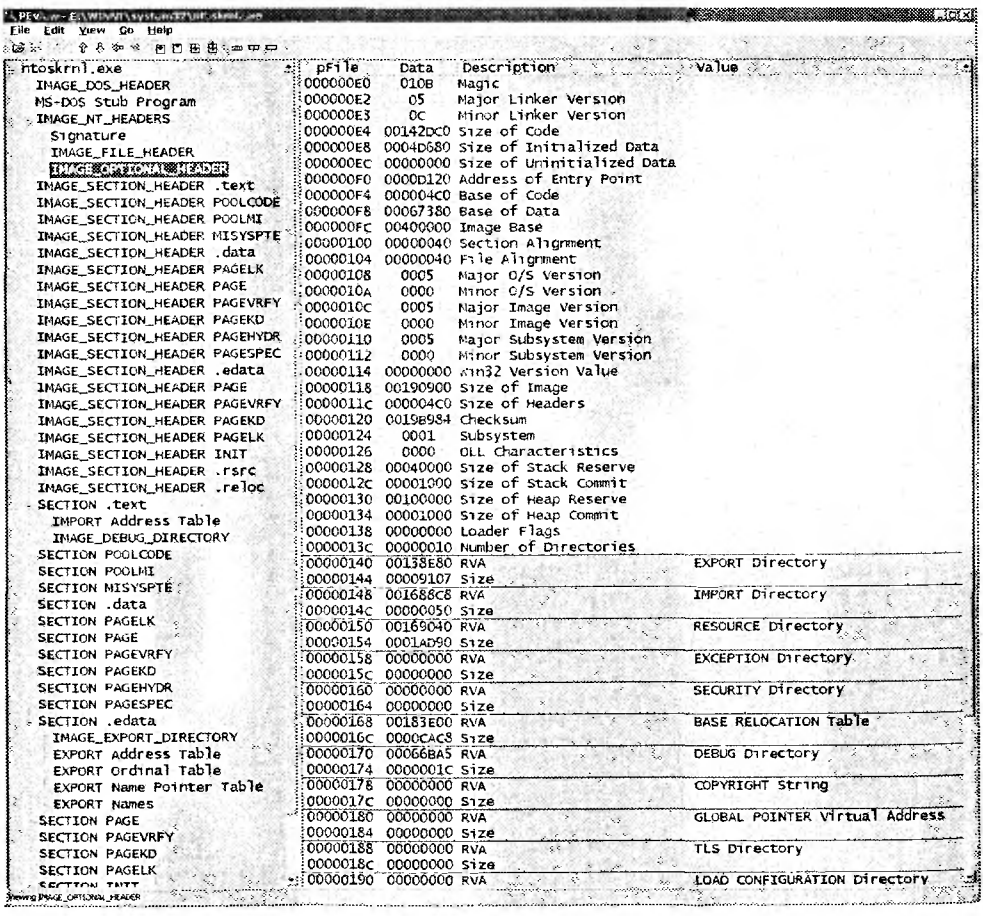

Рис. 1.9. Изучение внутренней структуры модуля ntoskrnl.exe при помощи утилиты PEview

Взглянув на панель инструментов программы Peview, вы обнаружите навигационные стрелки, позволяющие последовательно просматривать структуру файла (вертикальные стрелки), а также переходить от одного узла к другому в соответствии с историей просмотра (аналогично стрелкам Forward и Backward браузера Internet Explorer). Огромное количество команд, которые вы можете отдать программе PEview при помощи главного меню и панели инструментов, делает работу с этой программой настоящим удовольствием. Помимо исполняемых файлов приложений и файлов DLL утилита PEview позволяет просматривать логическое строение файлов некоторых других форматов, например файлов объектного кода, импортируемых библиотек, а также файлов символьных идентификаторов. Более подробно об утилите PEview можно узнать по адресу http://www.magma.ca/~wjr/. Это домашняя страничка автора программы Вэйна Дж. Рэдберна (Wayne J. Radburn).

Как я уже отмечал в предисловии, Вэйн занимается разработкой программного обеспечения Win32 на языке ассемблера (ASM). Да, да, это не только возможно, но и совсем несложно, достаточно обладать необходимыми для этого инструментами. На самом деле разрабатывать ассемблерные программы в среде Win32 значительно проще, чем в среде DOS или Windows 3.x, так как теперь вы можете воспользоваться преимуществами 32-битных процессорных инструкций. Вэйн активно поддерживает программирование на ассемблере в среде Win32, на его web-страничке вы найдете огромный объем исходного кода. В свое время я тоже был активным сторонником программирования на ассемблере, однако со временем я отказался от этой практики. Это произошло после того, как я обнаружил, что встроенный оптимизатор Microsoft Visual C в большинстве случаев формирует существенно более эффективный исполняемый код, чем это под силу человеку. Дело в том, что оптимизатор пользуется многими хитрыми приемами, применять которые программистам на ассемблере строго не рекомендуется, так как при их использовании ассемблерный код будет малопонятным, нечитаемым, запутанным и плохо модифицируемым. Но в то время, когда я активно занимался поддержкой ассемблерного программирования, мною был разработан чрезвычайно полезный пакет под названием Win32 Assembly Language Kit (WALK32), который является свободно распространяемым дополнением к Microsoft Macro Assembler (MASM). Получить WALK32 можно на моей домашней web-страничке по адресу http://www.orgon.com/pub/asm/. Вы должны скопировать к себе на жесткий диск все файлы, содержащие walk в своих именах. Приступая к работе с WALK32, имейте в виду, что я прекратил дальнейшую работу над этим пакетом и больше не обновляю и не поддерживаю его.

Отладочные интерфейсы Windows 2000

Отладчик Kernel Debugger — это чрезвычайно мощный инструмент исследования ядра операционной системы, однако его пользовательский интерфейс оставляет желать лучшего и часто вы оказываетесь в ситуации, когда требуется использовать более мощные, но не поддерживаемые отладчиком команды. К счастью, компания Microsoft добавила в Windows 2000 два полностью документированных отладочных интерфейса, используя которые вы можете добавить механизмы, связанные с отладкой ОС, в ваши собственные приложения. Отладочные интерфейсы Windows 2000

во многом не идеальны, однако они официально документированы компанией Microsoft. В данном разделе главы я представлю краткое описание этих интерфейсов, а также продемонстрирую, как возможности этих интерфейсов можно применить на практике.

psapi.dll, imagehlp.dll и dbghelp.dll

В течение долгого времени операционную систему Windows NT часто критиковали, указывая на отсутствие в ней интерфейса, подобного интерфейсу ToolHelp32 операционной системы Windows 95. Очевидно, что многие критики не знали о том, что Windows NT 4.0 имеет свой собственный альтернативный отладочный интерфейс, поддержка которого осуществляется исполняемым кодом модуля psapi.dll, распространяемого в составе пакета Win32 SDK. Помимо psapi.dll работа официально документированных отладочных интерфейсов Windows NT и Windows 2000 поддерживается модулями imagehlp.dll и dbghlp.dll. Таким образом, эти три модуля являются провайдерами отладочных функций в среде Win32. Аббревиатура PSAPI расшифровывается как Process Status Application Programming Interface — прикладной программный интерфейс состояния процесса. В состав этого интерфейса входят 14 функций, предоставляющих прикладным программам разнообразную системную информацию о драйверах устройств, процессах, использовании памяти, программных модулях, рабочих наборах и отображенных на память файлах. Модуль psapi.dll поддерживает работу как с ANSI, так и с Unicode.

Две другие отладочные библиотеки imagehlp.dll и dbghelp.dll используются для выполнения других задач. Обе библиотеки экспортируют сходные наборы функций. Отличие между этими библиотеками состоит в том, что imagehlp.dll содержит в себе большее количество функций, в то время как dbghelp.dll можно распространять в составе собственных продуктов. Другими словами, компания Microsoft разрешает добавлять файл dbghelp.dll в комплект поставки приложений сторонних разработчиков. Обе библиотеки содержат большой набор функций для грамматического разбора, анализа и обработки файлов формата PE. На мой взгляд, самой важной особенностью этих библиотек является возможность извлечения символьных идентификаторов из соответствующих файлов, устанавливаемых вместе с отладчиком Kernel Debugger. Чтобы читатели смогли получить представление о возможностях обеих библиотек, я перечисляю функции, экспортируемые этими библиотеками, в табл. 1.1. В этой таблице указывается, какие из функций поддерживаются каждой из них.

Таблица 1.1. Сравнение библиотек imagehlp.dll и dbghelp.dll

Имя	imagehlp.dll	dbghelp.dll
BindImage		Нет
BindImageEx		Нет
CheckSumMappedFile		Нет
EnumerateLoadedModules		
EnumarateLoadedModules64		
ExtentionApiVersion	Нет	

продолжение

Таблица 1.1 (*продолжение*)

Имя	imagehlp.dll	dbghelp.dll
FindDebugInfoFile		
FindDebugInfoFileEx		
FindExecutableImage		
FindExecutableImageEx		
FindFileInSearchPath		
GetImageConfigInformation		Нет
GetImageUnusedHeaderBytes		Нет
GetTimestampForLoadedLibrary		
ImageAddCertificate		Нет
ImageDirectoryEntryToData		
ImageDirectoryEntryToDataEx		
ImageEnumerateCertificates		Нет
ImageGetCertificateData		Нет
ImageGetCertificateHeader		Нет
ImageGetDigestStream		Нет
ImagehlpApiVersion		
ImagehlpApiVersionEx		
ImageLoad		Нет
ImageNtHeader		
ImageRemoveCervificate		Нет
ImageRvaToSection		
ImageRvaToVa		
ImageUnload		Нет
MakeSureDirectoryPathExists		
MapAndLoad		Нет
MapDebugInformation		
MapFileAndCheckSumA		Нет
MapFileAndCheckSumW		Нет
ReBaseImage		Нет
ReBaseImage64		Нет
RemovePrivateCvSymbolic		Нет
RemovePrivateCvSymbolicEx		Нет
RemoveRelocations		Нет
SearchTreeForFile		
SetImageConfigInformation		Нет
SplitSymbols		Нет
StackWalk		
StackWalk64		
sym	Нет	
SymCleanup		
SymEnumerateModules		

Имя	imagehlp.dll	dbghelp.dll
SymEnumerateModules64		
SymEnumerateSymbols		
SymEnumerateSymbols64		
SymEnumerateSymbolsW		
SymFunctionTableAccess		
SymFunctionTableAccess64		
SymGetLineFromAddr		
SymGetLineFromAddr64		
SymGetLineFromName		
SymGetLineFromName64		
SymGetLineNext		
SymGetLineNext64		
SymGetLinePrev		
SymGetLinePrev64		
SymGetModuleBase		
SymGetModuleBase64		
SymGetModuleInfo		
SymGetModuleInfo64		
SymGetModuleInfoEx		
SymGetModuleInfoEx64		
SymGetModuleInfoW		
SymGetModuleInfoW64		
SymGetOptions		
SymGetSearchPath		
SymGetSymbolInfo		
SymGetSymbolInfo64		
SymGetSymFromAddr		
SymGetSymFromAddr64		
SymGetSymFromName		
SymGetSymFromName64		
SymGetSymNext		
SymGetSymNext64		
SymGetSymPrev		
SymGetSymPrev64		
SymInitialize		
SymLoadModule		
SymLoadModule64		
SymMatchFileName		
SymEnumerateSymbolsW64		
SymRegisterCallback		
SymRegisterCallback64		
SymRegisterFunctionEntryCallback		

продолжение

Таблица 1.1 (*продолжение*)

Имя	imagehlp.dll	dbghelp.dll
SymRegisterFunctionEntryCallback64		
SymSetOptions		
SymSetSearchPath		
SymUnDName		
SymUnDName64		
SymUnloadModule		
SymUnloadModule64		
TouchFileTimes		Нет
UnDecorateSymbolName		
UnMapAndLoad		Нет
UnmapDebugInformation		
UpdateDebugInfoFile		Нет
UpdateDebugInfoFileEx		Нет
WinDbgExtentionDllInit	Нет	

В следующих разделах я рассмотрю несколько фрагментов исходного кода, иллюстрирующих применение psapi.dll и imagehlp.dll для решения следующих задач:

- получение перечня всех компонентов и драйверов ядра;
- получение перечня всех работающих в системе процессов;
- получение перечня всех модулей, загруженных в виртуальном адресном пространстве текущего процесса;
- получение перечня всех символьных идентификаторов данного компонента.

Интерфейс модуля psapi.dll спроектирован не самым лучшим образом. Он обеспечивает минимум функциональности, хотя, на мой взгляд, этот интерфейс можно было бы легко расширить и сделать более удобным. Мало того, этот модуль получает от ядра достаточно много полезной информации, но большая часть этих данных остается недоступной для вызывающего процесса.

Функции модулей psapi.dll и imagehlp.dll не входят в состав стандартного Win32 API, поэтому соответствующие заголовочные файлы и импортируемые библиотеки не включаются автоматически во все проекты Visual C/C++. Чтобы получить доступ к функциям отладочных интерфейсов, необходимо внести в исходные файлы вашего проекта строки, приведенные в листинге 1.2. Первые две строки включают в проект необходимые заголовки, а две другие строки устанавливают динамическую связь с функциями, экспортируемыми из данных DLL.

Листинг 1.2. Добавление библиотек psapi.dll и imagehlp.dll в проект Visual C/C++

```
#include <imagehlp.h>
#include <psapi.h>

#pragma comment (linker, "/defaultlib:imagehlp.lib")
#pragma comment (linker, "/defaultlib:psapi.lib")
```

Исходный ход на компакт-диске

На прилагаемом к книге компакт-диске содержатся два проекта, основанных на использовании библиотек psapi.dll и imagehlp.dll. Первый проект — программа w2k_sym.exe — является утилитой извлечения символьных идентификаторов из произвольного файла символьных идентификаторов. При этом подразумевается, что ранее вы уже установили в системе файлы символьных идентификаторов (см. раздел «*Установка файлов символьных идентификаторов*»). Таблица идентификаторов может быть сортирована по имени, адресу, размеру данных, кроме того, получая сведения об идентификаторах, вы можете использовать фильтры. Помимо этого, программа w2k_sym.exe позволяет просматривать списки активных системных модулей/драйверов, запущенных процессов, а также модулей, загруженных в адресном пространстве текущего процесса. Другим рассматриваемым проектом является вспомогательная отладочная библиотека w2k_dbg.dll, которая предлагает набор удобных функций-оболочек, облегчающих работу с библиотеками psapi.dll и imagehlp.dll. Программа w2k_sym.exe целиком и полностью основана на использовании w2k_dbg.dll. Исходный код обоих проектов содержится в каталогах \src\w2k_dbg и \src\w2k_sym.

Функции, к которым обращается w2k_dbg.dll, перечислены в табл. 1.2. Символами A/W обозначается поддержка стандартов кодирования символов: A обозначает ANSI, а W обозначает Unicode. Как отмечалось ранее, библиотека psapi.dll поддерживает работу как с ANSI, так и с Unicode. К сожалению, библиотеки imagehlp.dll и dbghelp.dll не столь совершенны, и при обращении к некоторым функциям этих библиотек требуется использовать строки в 8-битном формате ANSI. Это обстоятельство подчас является источником дополнительных хлопот для программиста. Дело в том, что отладочные приложения Windows 2000, как правило, не предназначены для работы в среде Windows 9x, поэтому при их разработке было бы проще полностью отказаться от поддержки ANSI и работать только с символами Unicode. Однако если в рамках своего проекта вы планируете использовать imagehlp.dll, вы вынуждены либо использовать ANSI, либо время от времени конвертировать строки Unicode в ANSI и обратно. Лично я ненавижу работать с 8-битными строками в системе, которая обладает встроенной поддержкой 16-битных символов, поэтому я предпочитаю второй подход. Все функции, экспортируемые модулем w2k_dbg.dll и связанные со строками, работают только в стандарте Unicode, поэтому, работая с этой библиотекой, вы можете не беспокоиться о размере символов.

С другой стороны, библиотеки imagehlp.dll и dbghelp.dll обладают весьма любопытным свойством, отсутствующим в библиотеке psapi.dll: обе этих библиотеки включают в себя поддержку Win64 — стандарта разработки 64-битных приложений Windows. Стандарт Win64 наводит страх на подавляющее большинство разработчиков Windows, так как на текущий момент фактически никому не известно, насколько сложно будет адаптировать приложения Win32 для работы в среде Win64. Отладочные библиотеки imagehlp.dll и dbghelp.dll экспортируют 64-битные версии своих функций, и может быть, скоро настанет время, когда всем нам придется ими воспользоваться.

Таблица 1.2. Отладочные функции, к которым обращается библиотека w2k_dbg.dll

Имя	A/W	Библиотека
EnumDeviceDrivers		psapi.dll
EnumProcesses		psapi.dll
EnumProcessModules		psapi.dll
GetDeviceDriverFileName	A/W	psapi.dll
GetModuleFileNameEx	A/W	psapi.dll
GetModuleInformation		psapi.dll
ImageLoad	A	imagehlp.dll
ImageUnload		imagehlp.dll
SymCleanup		imagehlp.dll
SymEnumerateSymbols	A/W	imagehlp.dll
SymInitialize	A	imagehlp.dll
SymLoadModule	A	imagehlp.dll
SymUnloadModule		imagehlp.dll

В данной книге я не хочу слишком подробно рассматривать библиотеки psapi.dll и imagehlp.dll. Основной темой книги являются недокументированные интерфейсы Windows 2000, а функции обеих отладочных библиотек достаточно хорошо описаны в документации Platform SDK. Все же я не хочу полностью упускать их из виду, так как обе эти библиотеки тесно связаны с интерфейсом Windows 2000 Native API, речь о котором пойдет в главе 2. Кроме того, библиотека psapi.dll является отличным примером документированного интерфейса, вместо которого я рекомендую использовать недокументированный интерфейс. Отладочный интерфейс psapi.dll неудобен и груб, кроме того, в некоторых случаях его функции возвращают некорректную информацию. Если бы мне пришлось разрабатывать собственный коммерческий профессиональный программный продукт, связанный с отладкой, я бы ни в коем случае не стал бы использовать при его создании функции библиотеки psapi.dll. Ядро Windows 2000 предлагает набор более мощных, более гибких и более удобных отладочных функций, которые, к сожалению, являются недокументированными. К счастью, эти функции активно используются многими системными утилитами Microsoft, в связи с этим от версии к версии Windows NT они меняются очень незначительно. Конечно же, если вы используете этот недокументированный интерфейс, вы должны тщательно протестировать свой программный продукт для того, чтобы убедиться в его совместимости с каждой новой версией NT, однако у данного подхода значительно больше преимуществ, чем недостатков.

Большая часть рассматриваемых далее фрагментов исходного кода позаимствована из файла w2k_dbg.c и является частью библиотеки w2k_dbg.dll. Все файлы, необходимые для компиляции этой библиотеки, записаны на прилагаемом к книге компакт-диске в каталоге \src\w2k_dbg\. Данная библиотека является набором удобных функций, упрощающих работу с отладочными интерфейсами Windows 2000. Функции модуля w2k_dbg.dll позволяют получить массу полезной отладочной

информации в удобной форме. Возвращаемые функциями данные оформляются в виде связанных списков, которые в случае необходимости можно снабдить индексами, упрощающими поиск и сортировку. Функции, экспортируемые библиотекой w2k_dbg.dll, перечислены в табл. 1.3. Это достаточно обширный перечень, и подробное обсуждение каждой функции потребовало бы слишком много места. Вместо этого, для того чтобы понять методы использования данной библиотеки, я рекомендую читателям изучить исходный код приложения w2k_sym.exe (см. каталог \src\w2k_sym\w2k_sym.c на компакт-диске).

Таблица 1.3. Набор функций библиотеки w2k_dbg.dll

Имя функции	Описание
dbgBaseDriver	Определяет базовый адрес и размер драйвера исходя из его пути
dbgBaseModule	Возвращает базовый адрес и размер модуля DLL
dbgCrc32Block	Вычисляет CRC32 блока памяти
dbgCrc32Byte	Байтовое вычисление CRC32
dbgCrc32Start	Предварительная проверка CRC32
dbgCrc32Stop	Последующая проверка CRC32
dbgDriverAdd	Добавить запись о драйвере в список драйверов
dbgDriverAddresses	Возвращает список адресов драйверов (функция-оболочка для функции EnumDeviceDrivers())
dbgDriverIndex	Создает индексированный и при необходимости сортированный список драйверов
dbgDriverList	Создает плоский (неиндексированный) список драйверов
dbgFileClose	Закрывает файл на диске
dbgFileLoad	Загружает содержимое дискового файла в блок памяти
dbgFileNew	Создает новый дисковый файл
dbgFileOpen	Открывает существующий дисковый файл
dbgFileRoot	Возвращает смещение корневого токена пути к файлу
dbgFileSave	Сохраняет блок памяти в дисковом файле
dbgFileUnload	Освобождает блок памяти, созданный при помощи функции dbgFileLoad()
dbgIndexCompare	Сравнивает две записи, на которые указывает индекс (используется функцией dbgIndexSort())
dbgIndexCreate	Создает индекс указателей на основе списка объектов
dbgIndexCreateEx	Создает сортированный индекс указателей на основе списка объектов
dbgIndexDestroy	Освобождает память, занимаемую индексом и связанным с ним списком
dbgIndexDestroyEx	Освобождает память, занимаемую двумерным индексом и связанными с ним списками
dbgIndexList	Создает «плоскую» копию списка на основе индекса
dbgIndexListEx	Создает «плоскую» копию двумерного списка на основе индекса
dbgIndexReverse	Меняет порядок записей в списке, связанном с индексом, на обратный
dbgIndexSave	Сохраняет образ памяти, в которой располагается индексированный список, в дисковом файле
dbgIndexSaveEx	Сохраняет образ памяти, в которой располагается двумерный индексированный список, в дисковом файле

продолжение

Таблица 1.3 (*продолжение*)

Имя функции	Описание
dbgIndexSort	Сортирует записи списка, на которые ссылается индекс, в соответствии с адресом, размером, ID или именем
dbgListCreate	Создает пустой список
dbgListCreateEx	Создает пустой список с зарезервированным пространством памяти
dbgListDestroy	Освобождает память, занимаемую списком
dbgListFinish	Завершает последовательно построенный список и освобождает любую неиспользуемую память
dbgListIndex	Создает индекс указателей на основе списка
dbgListLoad	Создает список исходя из образа, сохраненного в файле
dbgListNext	Обновляет заголовок списка после добавления в список новой записи
dbgListResize	Резервирует память для дополнительных записей списка
dbgListSave	Сохраняет образ памяти, в которой хранится список, в дисковый файл
dbgMemoryAlign	Округляет байтовый счетчик до следующей 64-битной границы
dbgMemoryAlignEx	Округляет счетчик символов в строке до следующей 64-битной границы
dbgMemoryBase	Возвращает внутренний базовый адрес блока в пуле выделяемой памяти (heap memory block)
dbgMemoryBaseEx	Возвращает внутренний базовый адрес индивидуально помеченного блока в пуле выделяемой памяти (individually tagged heap memory block)
dbgMemoryCreate	Выделяет блок памяти из пула выделяемой памяти
dbgMemoryCreateEx	Выделяет индивидуально помеченный блок памяти из пула выделяемой памяти
dbgMemoryDestroy	Возвращает блок памяти в пул свободной памяти
dbgMemoryDestroyEx	Возвращает индивидуально помеченный блок памяти в пул свободной памяти
dbgMemoryReset	Переустанавливает статистику использования памяти
dbgMemoryResize	Изменяет размер блока памяти, выделенного из пула свободной памяти
dbgMemoryResizeEx	Изменяет размер индивидуально помеченного блока памяти, выделенного из пула свободной памяти
dbgMemoryStatus	Возвращает статистические данные об использовании памяти
dbgMemoryTrack	Обновляет статистические данные об использовании памяти
dbgModuleIndex	Создает индексированный и при необходимости сортированный подсписок модулей процесса
dbgModuleList	Формирует «плоский» подсписок модулей процесса
dbgPathDriver	Формирует спецификацию пути к драйверу по умолчанию
dbgPathFile	Возвращает смещение токена имени файла в составе пути к файлу
dbgPrivilageDebug	Запрашивает привилегию отладки для вызывающего процесса
dbgPrivilageSet	Запрашивает указанную привилегию для вызывающего процесса
dbgProcessAdd	Добавляет запись о процессе в список процессов
dbgProcessGuess	Предлагает отображаемое имя по умолчанию для анонимного системного процесса
dbgProcessIds	Возвращает массив идентификаторов процессов (функция-оболочка вызова EnumProcesses())

Имя функции	Описание
dbgProcessIndex	Создает индексированный и при желании сортированный список процессов
dbgProcessIndexEx	Создает двумерный индексированный и при желании сортированный список процессов/модулей
dbgProcessList	Создаёт «плоский» список процессов
dbgProcessModules	Возвращает список дескрипторов модулей процесса (функция-оболочка вызова EnumProcessModules())
dbgSizeDivide	Делит счетчик байтов на степень двойки. При желании выполняется округление вверх или вниз
dbgSizeKB	Преобразует количество байт в количество Кбайт. При желании результат округляется вверх или вниз
dbgSizeMB	Преобразует количество байт в количество Мбайт. При желании результат округляется вверх или вниз
dbgStringAnsi	Преобразует строку Unicode в строку ANSI
dbgStringDay	Возвращает имя дня недели исходя из порядкового номера дня недели
dbgStringMatch	Применяет фильтр к строке
dbgSymbolCallback	Добавляет запись о символьном идентификаторе в список символьных идентификаторов (вызывается функцией SymEnumerateSymbols())
dbgSymbolIndex	Создает индексированный и при желании сортированный список символьных идентификаторов
dbgSymbolList	Создает «плоский» список символьных идентификаторов
dbgSymbolLoad	Загружает таблицу символьных идентификаторов модуля
dbgSymbolLookup	Определяет имя и при желании смещение символьного идентификатора по заданному адресу
dbgSymbolUnload	Выгружает таблицу символьных идентификаторов модуля

Последовательный перебор системных модулей и драйверов

Библиотеку psapi.dll можно использовать для получения списка активных модулей ядра, в настоящее время располагающихся в памяти. Получение такого списка — совсем несложная задача. Входящая в состав этой библиотеки функция EnumDeviceDrivers() принимает массив слотов типа PVOID и заполняет его базовыми адресами активных драйверов режима ядра. В перечень включаются такие базовые модули ядра, как ntdll.dll, ntoskrnl.exe, win32k.sys, hal.dll и bootvid.dll. Полученные при помощи этой функции значения являются виртуальными адресами, по которым в памяти отображается содержимое соответствующих исполняемых файлов. Если при помощи отладчика или какой-либо иной отладочной утилиты вы посмотрите на содержимое нескольких первых ячеек памяти, расположенных по любому из этих адресов, вы без труда узнаете код заглушки DOS, начинающийся со знаменитых инициалов Марка Збиковски (Mark Zbikowski, MZ) и содержащий строку This program cannot be run in DOS mode или что-либо подобное. Пример обращения к функции EnumDeviceDrivers(), а также прототип этой функции содержатся в листинге 1.3.

Листинг 1.3. Получение перечня адресов системных модулей

```
BOOL WINAPI EnumDeviceDrivers (PVOID  *lpImageBase.
                               DWORD   cb
                               PWORD   lpcbNeeded):

PPVOID WINAPI dbgDriverAddresses (PDWORD pdCount)
 {
 DWORD     dSize:
 DWDRD     dCount = 0:
 PPVOID    ppList = NULL:

 dSize = SIZE_MINIMUM * sizeof (PVOID):

 while ((ppList = dbgMemoryCreate (dSize)) != NULL)
      {
      if (EnumDeviceDrivers (ppList, dSize, &dCount) &&
           (dCount < dSize))
           {
           dCount /= sizeof (PVOID):
           break:
           }
      dCount = 0:
      ppList = dbgMemoryDestroy (ppList):
      if ((dSize <<= 1) > (SIZE_MAXIMUM * sizeof (PVOID))) break:
      }
 if (pdCount != NULL) *pdCount = dCount:
 return ppList:
 }
```

Функция EnumDeviceDrivers() принимает три аргумента: указатель на массив, размер массива в байтах и указатель на переменную типа DWORD, в которой будет размещено количество байт, скопированных в массив. Обратите внимание на то, что второй аргумент должен соответствовать длине входного массива в байтах, а в переменной, на которую указывает третий аргумент, размещается количество байт, скопированных в этот массив. Таким образом, чтобы получить количество адресов, скопированных в массив, вы должны поделить это значение на величину sizeof (PVOID). Безусловно, функция EnumDeviceDrivers() знает о том, какое количество драйверов в настоящее время загружено в системе, однако при этом она не возвращает вызывающей программе информацию о том, каким должен быть размер входного массива для того, чтобы в нем уместились записи обо всех драйверах. Функция всего лишь возвращает количество байт, скопированных в массив. Если массив недостаточно велик для того, чтобы вместить информацию обо всех драйверах, не поместившиеся в него данные теряются. Таким образом, для того чтобы определить необходимый размер массива и получить информацию обо всех загруженных в системе драйверах, приходится использовать самую бесхитростную разновидность метода проб и ошибок. Эта процедура продемонстрирована в листинге 1.3. Предполагается, что если число скопированных функцией байт в точности равняется размеру входного массива, значит, массив недостаточно велик и некоторые данные в него не поместились. Изначально массив способен вместить в себя 256 записей (это значение представлено константой SIZE_MINIMUM). Как правило, этого достаточно, однако если количество драйверов в системе превышает 256, размер массива удва-

ивается. Так происходит до тех пор, пока либо в массиве не разместятся абсолютно все записи, возвращаемые функцией EnumDeviceDrivers(), либо требуемый размер массива не превысит максимального допустимого значения 65 536 (это значение представлено константой SIZE_MAXIMUM). Буфер в памяти выделяется и освобождается вспомогательными функциями dbgMemoryCreate() и dbgMemoryDestroy(), которые являются удобными функциями-оболочками стандартных функций Win32 с именами LocalAlloc() и LocalFree().

В листинге 1.4 показана одна из возможных реализаций функции EnumDeviceDrivers(). Представленная здесь реализация ни в коем случае не является оригинальным исходным кодом модуля psapi.dll. Это всего лишь случайная последовательность символов, которую можно откомпилировать при помощи компилятора C, и в результате, как я полагаю, вы получите бинарный код, эквивалентный бинарному коду функции EnumDeviceDrivers() модуля psapi.dll. Чтобы сделать код проще и понятнее, я убрал из него некоторые сбивающие с толку детали, присутствующие в изначальном коде функции. Например, мною были опущены операторы, относящиеся к структурной обработке исключений (SEH, Structured Exception Handling). В основе реализации, представленной в листинге 1.4, лежит обращение к вызову NtQuerySystemInformation(). Именно этот вызов выполняет всю наиболее сложную часть работы. Этот вызов является одним из особенно любимых мною вызовов Windows 2000, так как он позволяет получить доступ к содержимому многих важных внутренних структур этой операционной системы, таких как списки драйверов, процессов, потоков, дескрипторов, портов LPC и пр. Внутреннее строение, а также методы использования этой чрезвычайно мощной функции, и дополняющей ее функции NtSetSystemInformation() впервые были описаны в моей статье «Inside Windows NT System Data», опубликованной в ноябрьском номере журнала «Dr. Dobb's Journal» за 1999 год. Подробное описание обеих этих функций можно найти также в книге «Windows NT/2000 Native API Reference», написанной Гари Неббеттом (Gary Nebbett) и опубликованной издательством «Macmillan Technical» Publishing в 2000 году.

Листинг 1.4. Упрощенная реализация функции EnumDeviceDrivers()

```
BOOL WINAPI EnumDeviceDrivers (PVOID  *lpImageBase
                               DWORD   cb
                               DWORD  *lpcbNeeded)
 {
 SYSTEM_MODULE_INFORMATION_N(1)    smi;
 PSYSTEM_MODULE_INFORMATION        psmi;
 DWORD                             dSize, i;
 NTSTATUS                          ns;
 BOOL                              fOk = FALSE;

 ns = NtQuerySystemInformation(SystemModuleInformation,
                               &smi, sizeof (smi), NULL);

 if ((ns == STATUS_SUCCESS) ||
     (ns == STATUS_INFO_LENGTH_MISMATCH))
     {
     dSize = sizeof (SYSTEM_MODULE_INFORMATION) +
             (smi.dCount * sizeof (SYSTEM_MODULE));
```

продолжение

Листинг 1.4 (*продолжение*)

```
    if ((psmi = LocalAlloc (LMEM_FIXED, dSize)) != NULL)
        {
        ns = NtQuerySystemInformation (SystemModuleInformation,
                                       psmi, dSize, NULL);

        if (ns == STATUS_SUCCESS)
            {
            for (i = 0; (i < psmi->dCount) &&
                        (i < cb / sizeof (DWORD)); i++)
                {
                lpImageBase [i] = psmi->aModules [i].pImageBase;
                }
            *lpcbNeeded    = i * sizeof (DWORD);
            fOk                = TRUE;
            }
        LocalFree (psmi);

        if (!fOk) SetLastError (RtlNtStatusToDosError (ns));
        }
    }
else
    {
    SetLastError (rtlNtStatusToDosError (ns));
    }
return fDk;
}
```

Не следует уделять слишком много внимания особенностям реализации EnumDeviceDrivers(), представленной в листинге 1.4. Я добавил в книгу этот фрагмент кода лишь для того, чтобы продемонстрировать вам, что на использовании функции NtQuerySystemInformation() основана фактически вся библиотека psapi.dll. Также я хочу показать вам, что достаточно мощная функция NtQuerySystemInformation() используется данной библиотекой на удивление бездарно. Обратите внимание на то, как именно происходит обработка данных, возвращаемых функцией NtQuerySystemInformation(). Во втором по счету обращении к этой функции указывается информационный класс SystemModuleInformation, в результате чего функция возвращает полный список загруженных в системе драйверов. После этого производится последовательный перебор всех элементов массива, полученного в результате обращения к функции NtQuerySystemInformation(). Из каждого элемента этого массива извлекается базовый адрес модуля (поле pImageBase). Полученные таким образом адреса заносятся в результирующий массив указателей, возвращаемый вызывающему процессу (массив lpImageBase[]). Такая логика выглядит оправданной, но при этом возникает вопрос: «Какие данные помимо базовых адресов модулей содержатся в массиве, получаемом в результате обращения к функции NtQuerySystemInformation()?» Структура этих данных не документирована, однако на текущий момент я могу уверенно заявить, что помимо базовых адресов в составе возвращаемого ей массива функция NtQuerySystemInformation() передает вызывающему процессу такую бесценную информацию, как размеры модулей в памяти, пути и имена исполняемых файлов, счетчики загрузки, а также некоторые флаги. Сведения, возвращаемые функцией NtQuerySystemInformation(), включают в себя даже смещение имени фай-

ла в составе пути к модулю! И что же делает функция EnumDeviceDrivers()? Она безжалостно отбрасывает в сторону всю эту чрезвычайно полезную информацию и возвращает вызвавшему процессу только лишь базовые адреса модулей в памяти.

Не лучшим образом организованы и другие функции библиотеки, предназначенные для получения дополнительной информации о модулях исходя из их базовых адресов. Попытайтесь предположить, например, что происходит внутри функции GetDeviceDriverFileName(), которая предназначена для получения пути к файлу модуля, исходя из базового адреса этого модуля в памяти. Изучив код этой функции, можно обнаружить, что ее внутреннее строение сильно напоминает содержимое листинга 1.4. Вначале функция GetDeviceDriverFileName() обращается к вызову NtQuerySystemInformation() и с ее помощью получает полный список загруженных в системе драйверов. После этого она последовательно перебирает элементы этого списка в поисках совпадающего базового адреса. Обнаружив модуль с искомым базовым адресом, функция GetDeviceDriverFileName() извлекает из соответствующего элемента списка файловое имя модуля и копирует его в выходной буфер, предоставленный вызвавшим процессом. Ошарашивающая эффективность, неправда ли?! Почему нельзя было добавить соответствующий код в функцию EnumDeviceDrivers()? В этом случае все необходимые сведения о загруженных модулях можно было бы получить, выполнив просмотр массива, возвращаемого функцией NtQuerySystemInformation(), только один раз. Использование разных функций psapi.dll для получения различных сведений о загруженных драйверах не только малоэффективно, но и не вполне корректно. Представьте себе, что после обращения к функции EnumDeviceDrivers(), но перед обращением к функции GetDeviceDriverFileName() один из драйверов был выгружен из памяти. В этом случае одна из записей массива, полученного в результате обращения к EnumDeviceDrivers(), окажется некорректной и функция GetDeviceDriverFileName() не сможет определить имя драйвера, так как соответствующий указатель будет указывать куда угодно, но не на один из загруженных драйверов. Для меня остается загадкой, почему Microsoft занимается распространением столь безалаберно сделанной библиотеки.

Последовательный перебор активных процессов

Еще одной задачей, которую можно решить при помощи функций библиотеки psapi.dll, является получение сведений о процессах, работающих в системе. Для этой цели служит функция EnumProcesses(). Эта функция работает приблизительно также, как и функция EnumDeviceDrivers(), однако вместо списка виртуальных адресов данная функция возвращает список идентификаторов процессов (Process ID). Точно так же, как и EnumDeviceDrivers(), функция EnumProcesses() не позволяет заранее определить требуемый размер входного массива, поэтому, как и в предыдущем разделе, для получения полного списка процессов приходится использовать простой метод проб и ошибок. Процедура получения сведений о работающих в системе процессах продемонстрирована в листинге 1.5. Можно заметить, что содержащийся в листинге код фактически идентичен коду листинга 1.3. Отличаются только имена переменных и типов.

Листинг 1.5. Получение перечня идентификаторов процессов

```
BOOL WINAPI EnumProcess (DWORD  *lpidProcess.
                         DWORD   cd.
                         DWORD  *lpcbNeeded);

PDWORD WINAPI dbgProcessIds (PDWORD pdCount)
 {
 DWORD          dSize;
 DWORD          dCount = 0;
 PDWORD         pdList = NULL;

 dSize = SIZE_MINIMUM * sizeof (DWORD);

 while ((pdList = dbgMemoryCreate (dSize)) != NULL)
      {
      if (EnumProcesses (pdList, dSize, &dCount) &&
           (dCount < dSize))
           {
           dCount /= sizeof (DWDRD);
           break;
           }
      dCount = 0;
      pdList = dbgMemoryDestroy (pdList);
      if ((dSize <<= 1) > (SIZE_MAXIMUM * sizeof (DWORD))) break;
      }
 if (pdCount != NULL) *pdCount = dCount;
 return pdList;
 }
```

Идентификатор процесса (Process ID, PID) — это глобальная цифровая метка, уникально идентифицирующая процесс в рамках всей системы. Идентификаторы процессов и потоков извлекаются из одного и того же набора чисел. Нумерация начинается с нуля. Нулевым идентификатором обладает так называемый процесс холостого хода (Idle Process). В любой момент времени в системе не существует двух процессов или потоков, обладающих одним и тем же идентификатором PID, однако после того, как процесс завершает работу, используемые его потоками и им самим идентификаторы PID освобождаются и система может присвоить их другим процессам и потокам. Другими словами, идентификатор некоторого процесса, полученный в момент времени X, чуть позже, в момент времени Y, может соответствовать совершенно другому процессу. Мало того, в момент, когда вы попытаетесь его использовать, идентификатор может оказаться неопределенным, а кроме того, он может быть переназначен потоку. Таким образом, список идентификаторов PID, полученный в результате обращения к функции EnumProcesses(), на самом деле может не соответствовать текущему положению дел в системе. Недостатки этой функции становятся еще более очевидными, если изучить ее реализацию. Вероятная реализация этой функции содержится в листинге 1.6. Как и EnumDeviceDrivers(), для получения необходимой системной информации функция EnumProcesses() обращается к вызову NtQuerySystemInformation(), однако вместо SystemModuleInformation вызову передается информационный класс SystemProcessInformation. Обратите внимание на цикл в середине листинга 1.6, в котором массив lpidProcess[] заполняется данными, извлекаемыми из структуры SYSTEM_

PROCESS_INFORMATION. В лучших традициях компании Microsoft строение этой структуры не документируется.

Листинг 1.6. Простая реализация функции EnumProcesses()

```
BOOL WINAPI EnumProcesses (PDWORD   lpidProcess,
                           DWORD    cb,
                           PDWORD   lpcbNeeded)
 {
 PSYSTEM_PROCESS_INFORMATION  pspi, pspiNext;
 DWORD                        dSize, i;
 NTSTATUS                     ns;
 BOOL                         fOk = FALSE;

 for (dSize = 0x8000;
      ((pspi = LocalAlloc (LMEM_FIXED, dSize)) != NULL);
      dSize += 0x8000
      {
      ns = NtQuerySystemInformation (SystemProcessInformation,
                                     pspi, dSize, NULL);

      if (ns == STATUS_SUCCESS)
           {
           pspiNext = pspi;

           for (i = 0; i < cb / sizeof (DWORD); i++)
                {
                lpidProcess [i] = pspiNext->dUniqueProcessId;

                pspiNext = (PSYSTEM_PROCESS_INFORMATION)
                           ((PBYTE) pspiNext + pspiNext->dNext);
                }
           *lpcbNeeded    = i * sizeof (DWORD);
           fOk            = TRUE;
           }
      LocalFree (pspi);

      if (fOk || (ns != STATUS_INFO_LENGTH_MISMATCH))
           {
           if (!fOk) SetLastError (RtlNtStatusToDosError (ns));
           break;
           }
      }
 return fOk;
 }
```

Мы с вами уже видели, насколько пренебрежительно функция EnumDeviceDrivers() относится к данным, получаемым в результате обращения к вызову NtQuerySystemInformation(). Большая часть этих чрезвычайно полезных данных просто отбрасывается в сторону. Можно предположить, что функция EnumProcesses() поступает аналогичным образом. На самом деле функция EnumProcesses() еще более расточительна; она возвращает вызывающему процессу лишь мизерную долю тех обширных сведений о потоках и процессах, которые можно получить при помощи вызова NtQuerySystemInformation(). Судите сами: пока я пишу этот текст, на моем компьютере выполняется 37 процессов, при этом в результате своего выполнения

функция NtQuerySystemInformation() возвращает блок данных размером 24 488 байт, однако получив в свое распоряжение этот блок, функция EnumProcesses() извлекает из него лишь идентификаторы PID всех работающих в системе процессов, заносит все эти идентификаторы в массив и возвращает его вызывающему процессу. Размер результирующего массива составляет всего 148 байт. То есть все остальные 24 340 байт ценнейших сведений просто теряются!

Когда я ознакомился с внутренним строением функции EnumDeviceDrivers(), я несколько расстроился, однако когда закончил анализ внутреннего строения функции EnumProcesses(), мое сердце было окончательно разбито. Теперь, когда вы знакомы с тем, насколько упрощенный подход используется функциями модуля psapi.dll, вы, должно быть, поймете, почему я рекомендую использовать недокументированные функции API вместо документированных. Надеюсь приведенное здесь описание функций EnumDeviceDrivers() и EnumProcesses() будет для вас весомым аргументом в пользу применения вместо этих примитивных, но официально документированных функций недокументированного вызова NtQuerySystemInformation(). Зачем надо использовать менее эффективные и менее надежные функции вместо того, чтобы напрямую обратиться к вызову NtQuerySystemInformation() и получить всю интересующую вас системную информацию, так сказать, из первых рук и в более удобной форме? Хочу также отметить, что очень многие административные утилиты Microsoft вместо функций библиотеки psapi.dll используют вызов NtQuerySystemInformation(), почему бы не применить аналогичный подход в своих собственных продуктах?

Последовательный перебор модулей процесса

Определив идентификатор PID интересующего вас процесса, вы можете захотеть получить перечень модулей, загруженных в виртуальное адресное пространство этого процесса. Для получения такого списка можно использовать функцию EnumProcessModules(), которая также входит в состав библиотеки psapi.dll. В отличие от функций EnumDeviceDrivers() и EnumProcesses() функция EnumProcessModules() принимает четыре аргумента (прототип функции содержится в начале листинга 1.7). Дело в том, что функции EnumDeviceDrivers() и EnumProcesses() возвращают списки, являющиеся глобальными для всей системы, а функция EnumProcessModules() возвращает список, соответствующий конкретному процессу. Поэтому при обращении к функции этот процесс должен быть уникально идентифицирован при помощи дополнительного четвертого аргумента. В данном случае вместо PID для идентификации процесса используется дескриптор, то есть значение типа HANDLE. Чтобы получить дескриптор процесса, зная PID этого процесса, необходимо воспользоваться функцией OpenProcess().

Листинг 1.7. Получение списка модулей процесса

```
BOOL WINAPI EnumProcessModules (HANDLE     hProcess,
                                HMODULE  *lphModule,
                                DWORD      cb,
                                DWORD    *lpcbNeeded);
```

```
PHMODULE WINAPI dbgProcessModules  (HANDLE  hProcess,
                                    PDWORD  pdCount)
 {
 DWORD          dSize;
 DWORD          dCount = 0;
 PHMODULE  phList = NULL;

 if (hProcess != NULL)
      {
      dSize = SIZE_MINIMUM * sizeof (HMODULE);

      while ((phList = dbgMemoryCreate (dSize)) != NULL)
           {
           if (EnumProcessModules (hProcess, phList, dSize,
                                   &dCount))
                {
                if (dCount <= dSize)
                     {
                     dCount /= sizeof (HMODULE);
                     break;
                     }
                }
           else
                {
                dCount = 0;
                }
           phList = dbgMemoryDestroy (phList);
           if (!(dSize = dCount)) break;
           }
      }
 if (pdCount != NULL) *pdCount = dCount;
 return phList;
 }
```

В качестве результата своей работы функция EnumProcessModules() возвращает массив дескрипторов модулей процесса. Дескриптор модуля — это значение типа HMODULE. В Windows 2000 значение типа HMODULE — это просто базовый адрес модуля в памяти. В заголовочном файле windef.h, входящем в состав пакета Platform SDK, тип HMODULE определяется как псевдоним типа HINSTANCE, который, в свою очередь, совпадает с типом HANDLE. Возможно, разработчики из Microsoft выбрали подобное определение типов для того, чтобы подчеркнуть, что дескриптор модуля — это закрытая величина и о ее смысле не следует делать никаких предположений. На самом деле значение типа HMODULE не является дескриптором в строгом смысле этого слова. Как правило, под дескриптором понимают индекс в системной таблице, из которой извлекаются свойства объекта. Одному объекту могут соответствовать несколько дескрипторов. Каждый раз, когда система выделяет дескриптор, она увеличивает на единицу относящийся к объекту счетчик дескрипторов. Экземпляр объекта не может быть удален из памяти до того момента, пока все дескрипторы объекта не будут освобождены. Для освобождения дескриптора следует использовать стандартную функцию API под названием CloseHandle(). В контексте Native API эквивалентом этой функции является функция NtClose(). Однако в отличие от обычных дескрипторов дескрипторы модулей HMODULE освобождать *не* требуется.

Следует также учитывать, что корректность дескриптора модуля не гарантируется. В описании функции GetModuleHandle(), входящем в состав документации Platform SDK, говорится, что при разработке многопоточных приложений необходимо предпринять особенные меры, так как один из потоков может выгрузить модуль из памяти и тем самым сделать бессмысленным соответствующее значение типа HMODULE, используемое другим потоком. Это соображение имеет силу также и в многозадачной рабочей среде, в которой одно приложение (например, отладчик) намерено использовать дескриптор модуля, принадлежащий другому приложению. Принимая во внимание все это, можно прийти к выводу, что дескриптор модуля является фактически бесполезным значением. Однако на самом деле существуют две ситуации, в которых значение типа HMODULE остается корректным в течение достаточно долгого времени:

1. Значение типа HMODULE, полученное в результате обращения к вызовам LoadLibrary() или LoadLibraryEx(), остается корректным до того момента, пока процесс не обратится к вызову FreeLibrary(). Это связано с тем, что данные функции используют счетчик обращений к модулю, благодаря чему неожиданная выгрузка модуля исключается даже в многопоточной рабочей среде.
2. Значение HMODULE, полученное от другого процесса, остается корректным в случае, если оно соответствует модулю, постоянно загруженному в память. Например, все компоненты ядра Windows 2000 (исключая драйверы режима ядра) отображаются в виртуальные адресные пространства разных процессов по одним и тем же виртуальным адресам. Эти компоненты остаются по неизменным адресам в течение всего периода функционирования процесса.

К сожалению, ни одну из этих ситуаций нельзя отнести к дескрипторам модулей, возвращаемым функцией EnumProcessModules(), по крайней мере в большинстве случаев. Значения HMODULE, копируемые этой функцией в буфер вызывающего процесса, являются базовыми адресами модулей, находившихся в памяти в момент обращения к этой функции. Долей секунды позже конфигурация загруженных в рамках процесса модулей может измениться, так как любой из потоков процесса может обратиться к функции FreeLibrary() и выгрузить один из модулей из памяти. В результате соответствующий этому модулю дескриптор перестанет быть корректным. Мало того, если сразу же после этого произойдет обращение к функции LoadLibrary(), на место старого модуля по тому же базовому адресу может быть загружен совершенно другой модуль. Данная проблема сильно напоминает уже описанные ранее проблемы, с которыми приходится иметь дело при использовании функций EnumDeviceDrivers() и EnumProcesses(). Напомню, что первая из них возвращает массив указателей на драйверы ядра, а вторая — массив идентификаторов процессов, работающих в системе, однако оба этих массива в момент обращения к ним могут не соответствовать действительному положению дел в системе. Можно ли обойти эту проблему? Ответ на этот вопрос положителен, однако для решения задачи необходимо использовать недокументированные вызовы Windows 2000. Именно к этим вызовам обращаются функции библиотеки psapi.dll для того, чтобы получить необходимую системную информацию. В отличие от несовершенных функций библиотеки psapi.dll используемые ею недокументированные вызовы

возвращают вызывающему процессу более полный снимок системных объектов, включая все свойства этих объектов, которые могут представлять интерес. Таким образом, надобность в обращении к каким-либо другим функциям в более позднее время для получения дополнительных сведений об объектах отпадает. На мой взгляд, библиотека psapi.dll спроектирована из рук вон плохо, так как ее разработчики совершенно упустили из виду вопросы обеспечения целостности данных. Именно поэтому я не стал бы использовать эту библиотеку при разработке своего собственного профессионального отладочного приложения.

Функция EnumProcessModules() более удобна в использовании, чем ее предшественницы — функции EnumDeviceDrivers() и EnumProcesses(), так как эта функция сообщает вызывающему процессу, сколько именно байт не вместилось в предоставленный вызывающим процессом массив. Обратите внимание на то, что в листинге 1.7 отсутствует цикл, в котором буфер вывода увеличивается в размерах до тех пор, пока в нем не уместятся все выводимые функцией данные. Все же в данном случае без метода проб и ошибок не обойтись. Дело в том, что требуемый размер буфера, полученный в результате обращения к функции EnumProcessModules(), может перестать соответствовать действительности при следующем обращении к этой функции, так как между этими обращениями процесс может загрузить в свою память еще один модуль. Чтобы решить проблему, код листинга 1.7 продолжает обращаться к функции EnumProcessModules() до тех пор, пока либо требуемый размер буфера не станет меньшим или равным текущему размеру буфера, либо не произойдет ошибка.

Я не буду рассматривать исходный код вероятной реализации функции EnumProcessModules(), так как эта функция устроена несколько сложнее, чем функции EnumDeviceDrivers() и EnumProcesses(), кроме того, ее работа связана с несколькими недокументированными структурами данных. Для получения необходимой системной информации функция EnumProcessModules() обращается к недокументированному системному вызову NtQueryInformationProcess(). В результате обращения к этому вызову определяется адрес блока окружения целевого процесса (Process Environment Block, PEB), в котором, в свою очередь, содержится указатель на список сведений о модулях процесса. Ни блок PEB, ни список модулей целевого процесса не видимы из адресного пространства процесса, обращающегося к функции EnumProcessModules(). Чтобы прочитать информацию из адресного пространства другого процесса, используется документированная функция ReadProcessMemory(), входящая в состав стандартного интерфейса Win32 API. Внутреннее строение структуры PEB обсуждается позже — в главе 7. Кроме того, определение этой структуры содержится в приложении B.

Изменение привилегий процесса

Должно быть, вы помните, что при обращении к функции EnumProcessModules() требуется передать этой функции дескриптор процесса. Как правило, изначально вы обладаете идентификатором интересующего вас процесса, возможно, этот идентификатор является одним из идентификаторов, извлеченных из массива, полученного при обращении к функции EnumProcesses() библиотеки psapi.dll. Чтобы получить дескриптор процесса с известным PID, необходимо обратиться к функции

OpenProcess(), входящей в состав стандартного интерфейса Win32 API. В качестве первого аргумента эта функция принимает битовую маску флагов доступа. Например, если вы хотите получить дескриптор процесса с наиболее широкими правами на доступ к этому процессу, то, предположив, что идентификатор процесса хранится в переменной dId типа DWORD, вы должны выполнить следующее обращение: OpenProcess (PROCESS_ALL_ACCESS, FALSE, dId). Если попробовать выполнить подобное обращение в отношении процессов с небольшими по значению идентификаторами PID, функция OpenProcess() не сможет предоставить вам дескриптор и вместо этого вернет код ошибки. В чем дело? Доступ к процессам с небольшими по значению идентификаторами PID блокируется механизмами безопасности операционной системы. Процессы с небольшими PID являются жизненно важными системными службами, обеспечивающими функционирование всей операционной системы. Обычным пользовательским процессам запрещено выполнять любые возможные операции в отношении системных служб. В частности, в надежной, хорошо защищенной системе обычный пользовательский процесс не должен обладать возможностью остановить работу любого другого функционирующего процесса. Если какое-либо приложение остановит работу важной системной службы, функционирование всей системы будет нарушено. По этой причине определенные действия в отношении системных процессов могут выполнятся только процессами, обладающими необходимым для этого набором привилегий.

Уровень привилегий приложения можно повысить. В частности, отладчик для выполнения своих функций должен обладать самыми широкими полномочиями в отношении всех процессов системы. Изменение уровня привилегий процесса выполняется при помощи следующих действий:

1. Прежде всего необходимо открыть токен доступа процесса. Для этого следует воспользоваться функцией OpenProcessToken(), входящей в состав библиотеки advapi32.dll стандартного интерфейса Win32.
2. После успешного выполнения предыдущей процедуры необходимо подготовить структуру TOKEN_PRIVILEGES. в которой необходимо разместить информацию о требуемом уровне привилегий. Эта задача решается при помощи функции LookupPrivilageValue(), которая также входит в состав модуля advapi32.dll. Привилегия идентифицируется при помощи символьного имени. В заголовочном файле winnt.h, входящем в состав пакета Platform SDK, определяется 27 имен привилегий. Каждой из привилегий ставится в соответствие символьный идентификатор. Например, привилегии отладки соответствует символьный идентификатор SE_DEBUG_NAME, который соответствует строке «SeDebugPrivilege».
3. После выполнения первых двух шагов следует обратиться к функции AdjustTokenPrivilages(), которой следует передать дескриптор токена процесса, а также инициализированную структуру TOKEN_PRIVILEGES. Эта функция также экспортируется модулем advapi32.dll.

Если обращение к функции OpenProcessToken() завершилось успехом, не забудьте в дальнейшем, после выполнения всех необходимых процедур, закрыть дес-

криптор токена процесса. В состав библиотеки w2k_dbg.dll входит функция dbgPrivilegeSet(), которая выполняет все описанные действия и изменяет уровень привилегий процесса. Исходный код этой функции приведен в листинге 1.8. В конце листинга приведен текст еще одной функции модуля w2k_dbg.dll. Функция dbgPrivilegeDebug() является простой, но чрезвычайно удобной фунцией-оболочкой вызова dbgPrivilegeSet(). Функция dbgPrivilageDebug() запрашивает у системы привилегию отладки. Кстати говоря, аналогичный прием использует утилита kill.exe из комплекта добавлений Microsoft Windows NT Server Resource Kit. Для того чтобы убрать из памяти зависшую службу, утилита kill.exe должна обладать привилегией отладчика. Утилита kill.exe является незаменимой в ситуации, когда вы намерены перезапустить службу, не реагирующую на команды, и при этом не хотите осуществлять полную перезагрузку сервера. Каждый, кто работал с сервером Microsoft Internet Information Server (IIS) в Web, интранет или экстранет, хорошо знает, что данная утилита всегда должна быть под рукой, чтобы в любой момент вы смогли отдать команду kill inetinfo.exe.

Листинг 1.8. Изменение уровня привилегий процесса

```
BOOL WINAPI dbgPrivilegeSet (PWORD pwName)
 {
 HANDLE                  hToken;
 TOKEN_PRIVILEGES        tp;
 BOOL                    fOk = FALSE;

 if ((pwName != NULL)
     &&
     OpenProcessToken (GetCurrentProcess (),
                       TOKEN_ADJUST_PRIVILEGES,
                       &hToken))
     {
     if (LookupPrivilegeValue (NULL, pwName,
                               &tp.Privileges->Luid))
         {
         tp.Privileges=>Attributes      = SE_PRIVILEGE_ENABLED;
         tp.PrivilegeCount              = 1;

         fOk = AdjustTokenPrivileges (hToken, FALSE, &tp,
                                      0, NULL, NULL)
             &&
             (GetLastError () == ERROR_SUCCESS);
         }
     CloseHandle (hToken);
     }
 return fOk;
 }

//  ------------------------------------------------------------------

BOOL WINAPI dbgPrivilegeDebug (void)
 {
 return dbgPrivilegeSet (SE_DEBUG_NAME);
 }
```

Последовательный перебор идентификаторов

После того как мы бесцеремонно обругали библиотеку psapi.dll, давайте несколько сменим тон. Возможно psapi.dll реализована неудачно, но, с другой стороны, библиотека imagehlp.dll — настоящая жемчужина! Я познакомился с этим замечательным образцом программного обеспечения при поиске дополнительной информации о внутренней структуре файлов идентификаторов Windows 2000. Вышедшая три года назад статья Матта Питрека (Matt Pietrek, 1997b), одного из лучших мировых «взломщиков» Windows 2000, окончательно убедила меня в том, что знать структуру файлов идентификаторов абсолютно необходимо, раз уж imagehlp.dll готова ее показать. Я до сих пор в этом уверен. Фокус осуществляется функцией API SymEnumerateAymbols(), прототип которой показан в верхней половине листинга 1.9. С того времени я довольно много узнал о наиболее важных особенностях внутренней организации файлов идентификаторов Windows NT 4.0 и Windows 2000, поэтому библиотека imagehlp.dll больше мне нужна. Эта информация будет рассмотрена далее в этой главе.

Аргумент hProcess обычно представляет собой дескриптор вызываемого процесса, поэтому ему можно присвоить значение, возвращаемое GetCurrentProcess(). Заметьте, что GetCurrentProcess() возвращает не фактический дескриптор процесса, а константу 0xFFFFFFFF, называемую псевдодескриптором (pseudo handle). Это значение корректно для всех функций API, принимающих дескриптор процесса. Еще один псевдодескриптор, 0xFFFFFFFE, интерпретируется как дескриптор текущего потока. Он возвращается схожей функцией API GetCurrentThread().

Аргумент BaseOfDll определен как DWORD, хотя на самом деле это разновидность одного из типов HMODULE или HINSTATNCE. Полагаю, что Microsoft специально выбрала этот тип данных, подчеркивая тем самым, что это значение не обязано быть допустимой величиной типа HMODULE, хотя зачастую таковым является. SymEnumerateSymbols() вычисляет базовые адреса всех перебираемых идентификаторов относительно этого значения. Эту функцию можно без проблем использовать для запроса идентификаторов DLL, не загруженных в данный момент в адресное пространство процесса, поэтому значение BaseOfDll можно выбирать произвольно.

Листинг 1.9. SymEnumerateSymbols() и ее функция обратного вызова

```
BOOL  IMAGEAPI  SymEnumerateSymbols
          (IN HANDLE                      hrocess,
           IN DWORD                       BaseOfDll,
           IN PSYM_ENUMSYMBOLS_CALLBACK   Callback,
           IN PVOID                       UserContext);

typedef  BOOL  (CALLBACK  *PSYM_ENUMSYMBOLS_CALLBACK)
          (PTSTR              SymbolName,
          DWORD               SymbolAddress,
          PVOID               UserContext);
```

Аргумент Callback — это указатель на определяемую пользователем функцию обратного вызова, выполняющуюся для каждого идентификатора. Тип ее параметров показан в нижней части листинга 1.9. Функция обратного вызова принимает завершающуюся нулем строку с именем идентификатора, базовый адрес идентификато-

ра в соответствии с значением аргумента BaseOfDll функции SymEnumerateSymbols() и оценку размера элемента, которому присвоен идентификатор. SymbolName определена как PTSTR, поэтому ее фактический тип зависит от того, какая версия SymEnumerateSymbols(), ANSI или Unicode, была вызвана. В документации Platform SDK явно утверждается, что SymbolSize — это «наилучшее предполагаемое значение», которое может быть нулем. Если вам нужны только идентификаторы, ссылающиеся на настоящий код или данные, удобно отфильтровать эти специальные случаи.

UserContext представляет собой произвольный указатель, при помощи которого вызывающая программа может отслеживать последовательность перебора. Например, он может указывать на блок памяти, в котором расположена информация об идентификаторе. Этот указатель идентичен аргументу UserContext, который передается функции Callback. Функция обратного вызова может в любой момент прервать перебор, возвратив значение FALSE. Это действие обычно предпринимается при возникновении непоправимой ошибки или в том случае, если вызывающая программа получила ожидаемую информацию.

Листинг 1.10 демонстрирует типичную ситуацию, в которой применяется функция SymEnumerateSymbols(). Этот пример снова взят из w2k_dbg.dll. Для последовательного перебора идентификаторов определенного модуля нужно предпринять следующие действия:

1. Прежде всего необходимо вызвать функцию SymInitialize(), чтобы инициализировать дескриптор идентификатора. Прототип этой функции вместе с прототипами других обсуждаемых здесь функций показан в листинге 1.11. Параметр hProcess может быть дескриптором любого активного процесса в системе. Он используется для определения требуемого процесса отладчиками, предоставляющими информацию об идентификаторах нескольких процессов. Приложения, которым просто требуется последовательно перебрать идентификаторы в автономном режиме, могут передать значение GetCurrentProcess(). Выделенные функцией SymInitialize ресурсы должны быть затем освобождены вызовом SymCleanup().

Листинг 1.10. Создание списка идентификаторов

```
PDBG_LIST WINAPI dbgSymbolList (PWORD  pwPath,
                                PVOID  pBase)
    {
    PLOADED_IMAGE       pli;
    HANDLE              hProcess = GetCurrentProcess ();
    PDBG_LIST           pdl      = NULL;

    if ((pwPath != NULL) &&
        SymInitialize (hProcess, NULL, FALSE))
        {
        if ((pli = dbgSymbolLoad (pwPath, pBase, hProcess))
            != NULL)
            {
            if ((pdl = dbgListCreate ()) != NULL)
```

продолжение

Листинг 1.10 (*продолжение*)

```
                    {
                    SymEnumerateSymbols  (hProcess,
                                         (DWORD_PTR) pBase,
                                         dbgSymbolCallback,
                                         &pdl);
                    }
                dbgSymbolUnload (pli, pBase, hProcess);
                }
            SymCleanup (hProcess);
            }
        return dbgListFinish (pdl);
}
```

2. Для получения точной информации о модуле, для которого будет производиться последовательный перебор идентификаторов, в данный момент целесообразно вызвать функцию ImageLoad(). Обратите внимание на то, что эта функция присутствует только в imagehlp.dll, свободно распространяемый компонент dbghelp.dll ее не экспортирует. Функция ImageLoad() возвращает указатель на структуру LOADED_IMAGE, содержащую очень подробную информацию о загруженном модуле (листинг 1.11). Позже выделенная под эту структуру память должна быть освобождена при помощи ImageUnload().
3. Последнее действие, которое осталось предпринять перед вызовом SynEnumerateSymbols(), — это загрузить таблицу идентификаторов нужного модуля при помощи SymLoadModule(). Если перед этим вызывалась ImageLoad(), в качестве соответствующих аргументов могут быть переданы члены hFile и SizeOfImage возвращенной ImageLoad() структуры LOADED_IMAGE. Иначе необходимо установить hFile в NULL и SizeOfImage в ноль. Тогда SymLoadModule() попытается получить размер образа из файла идентификаторов, а этот размер не обязан быть точным. Позже таблица идентификаторов должна быть выгружена при помощи SymUnloadModule().

Листинг 1.11. Различные прототипы функций API из модуля imagehlp.dll

```
BOOL IMAGEAPI SymInitialize (HANDLE  hProcess,
                             PSTR    UserSearchPath,
                             BOOL    fInvadeProcess);

BOOL IMAGEAPI SymCleanUp (HANDLE hProcess);

DWORD IMAGEAPI SymLoadModule  (HANDLE  hProcess,
                               HANDLE  hFile,
                               PSTR    ImageName,
                               PSTR    ModuleName,
                               DWORD   BaseOfDll,
                               DWORD   SizeOfDll);

BOOL IMAGEAPI SymUnloadModule (HANDLE  hProcess,
                               DWORD   BaseOfDll);

PLOADED_IMAGE IMAGEAPI ImageLoad (PSTR  DllName,
                                  PSTR  DllPath);
```

```
BOOL  IMAGEAPI  ImageUnload
                    (PLOADED_IMAGE  LoadedImage);

typedef struct _LOADED_IMAGE
      {
      PSTR                    ModuleName;
      HANDLE                  hFile;
      PUCHAR                  MappedAddress;
      PIMAGE_NT_HEADERS       FileHeader;
      PIMAGE_SECTION_HEADER   LastRvaSection;
      ULONG                   NumberOfSections;
      PIMAGE_SECTION_HEADER   Sections;
      ULONG                   Characteristics;
      BOOLEAN                 fSystemImage;
      BOOLEAN                 fDOSImage;
      LIST_ENTRY              Links;
      ULONG                   SizeOfImage;
      }
      LOADED_IMAGE,  *PLOADED_IMAGE;
```

В листинге 1.10 сразу заметны функции SymInitialize(), SymEnumerateSymbols() и SymCleanup(). Не обращайте внимания на вызовы dbgListCreate() и dbgListFinish(), они относятся к тем функциям API из модуля w2k_dbg.dll, которые помогают создавать списки объектов в памяти. Остальные упомянутые выше ссылки на функции imagehlp.dll скрыты внутри функций API из модулей w2k_dbg.dll dbgSymbolLoad() и dbgSymbolUnload(), показанных в листинге 1.12. Обратите внимание: DbgSymbolLoad() обращается к dbgStringAnsi(), чтобы преобразовать строку с путем модуля из Unicode в ANSI, поскольку imagehlp.dll не экспортирует версию ImageLoad() для Unicode.

Листинг 1.12. Загрузка и выгрузка информации об идентификаторах

```
PLOADED_IMAGE WINAPI dbgSymbolLoad (PWORD    pwPath,
                                    PVOID    pBase,
                                    HANDLE   hProcess)
      {
      WORD              awPath [MAX_PATH];
      PBYTE             pbPath;
      DWORD             dPath;
      PLOADED_IMAGE     pli = NULL;

      if ((pbPath = dbgStringAnsi (pwPath, NULL)) != NULL)
           {
           if (((pli = ImageLoad (pbPath, NULL)) == NULL) &&
                (dPath = dbgPathDriver (pwPath, awPath,
                                        MAX_PATH)) &&
                (dPath < MAX_PATH))
                {
                dbgMemoryDestroy  (pbPath);

                if ((pbPath = dbgStringAnsi (awPath, NULL)) != NULL)
                     {
                     pli = ImageLoad (pbPath, NULL);
                     }
                }
```

продолжение

Листинг 1.12 (*продолжение*)

```
        if ((pli != NULL)
            &&
            (!SymLoadModule (hProcess, pli->hFile, pbPath,
                             NULL, (DWORD_PTR) pBase,
                             pli->SizeOfImage)))
            {
            ImageUnload (pli);
            pli = NULL;
            }
        dbgMemoryDestroy (pbPath);
        }
    return pli;
    }
//  ------------------------------------------------------

PLOADED_IMAGE WINAPI dbgSymbolUnload (PLOADED_IMAGE  pli,
                                      PVOID          pBase,
                                      HANDLE         hProcess)
    {
    if (pli != NULL)
        {
        SymUnloadModule (hProcess, (DWORD_PTR) pBase);
        ImageUnload     (pli);
        }
    return NULL;
    }

//  ------------------------------------------------------

PDBG_LIST WINAPI dbgSymbolList (PWORD  pwPath,
                                PVOID  pBase)
    {
    PLOADED_IMAGE      pli;
    HANDLE             hProcess = GetCurrentProcess ();
    PDBG_LIST          pdl      = NULL;

    if ((pwPath != NULL) &&
        SymInitialize (hProcess, NULL, FALSE))
        {
        if ((pli = dbgSymbolLoad (pwPath, pBase, hProcess)) != NULL)
            {
            if ((pdl = dbgListCreate ()) != NULL)
                {
                SymEnumerateSymbols  (hProcess,
                                      (DWORD_PTR) pBase,
                                      dbgSymbolCallback, &pdl);
                }
            dbgSymbolUnload (pli, pBase, hProcess);
            }
        SymCleanup (hProcess);
        }
    return dbgListFinish (pdl);
    }
```

ImageLoad() делает очень много, определяя расположение указанного модуля, даже если известно только его имя, без какой-либо информации о пути. У нее, однако, ничего не получится с драйверами режима ядра, расположенными в каталоге \winnt\system32\drivers, поскольку этот каталог не входит в список путей поиска системы. В этом случае dbgSymbolLoad() обращается за помощью к функции dbgPathDriver() и снова вызывает LoadImage(). dbgPathDriver() просто добавляет к указанному пути префикс «driver\», если путь состоит только из одного имени файла. Если один из вызовов ImageLoad() вернет допустимый указатель на структуру LOADED_IMAGE, dbgSymbolLoad() заканчивает работу, загружая таблицу идентификаторов модуля при помощи вызова SymLoadModule() и, в случае успеха, возвращая структуру LOADED_IMAGE.

В листинге 1.10 SymEnumerateSymbols() для обратных вызовов использует функцию dbgSymbolCallback() из w2k_dbg.dll. Здесь я не привожу ее исходный код, поскольку эта функция не имеет отношения к библиотеке imagehlp.dll. dbgSymbolCallback() берет переданную ей информацию об идентификаторах (см. определение PSYM_ENUMSYMBOLS_CALLBACK в листинге 1.9) и добавляет ее к блоку памяти, переданному ей через параметр-указатель UserContext. Хотя функции перечисления, индексации и сортировки из библиотеки w2k_dbg.dll сами по себе представляют интерес, они выходят за рамки материала данной книги. Для получения дополнительной информации обращайтесь к файлам исходного кода w2k_dbg.dll и w2k_sym.exe, расположенным на компакт-диске.

Браузер идентификаторов Windows 2000

Программа w2k_sym.exe реализует клиентское приложение-пример для библиотеки w2k_dbg.dll, выполняющееся в консольном режиме Win32. При вызове без параметров программа выводит справочную информацию, показанную в примере 1.5. Как видно, программа называется Windows 2000 Symbol Browser (браузер идентификаторов Windows 2000). Управление осуществляется при помощи ключей командной строки, основные четыре параметра следующие: /p (вывести список процессов), /m (вывести список модулей процесса /d (вывести список драйверов и системных модулей) или путь к модулю, для которого запрашивается символьная информация. Поведение по умолчанию можно изменить при помощи различных ключей режима отображения, сортировки и фильтрации. Например, для того чтобы вывести список идентификаторов ntoskrnl.exe в отсортированном по имени виде, нужно выполнить команду w2k_sym /n/v ntoskrnl.exe. Ключ /n задает режим сортировки по имени, а /v определяет подробное (verbose) отображение информации, без него будут выведены только итоги.

Пример 1.5. Справка по командам w2k_sym.exe

```
// w2k_sym.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com
```

продолжение

Пример 1.5 (*продолжение*)

```
Usage: w2k_sym { <mode> [ /f | /F <filter> ] <operation> }

<mode> is a series of options for the next <operation>:

          /a : sort by address
          /s : sort by size
          /i : sort by ID (process/module lists only)
          /n : sort by name
          /c : sort by name (case-sensitive)
          /r : reverse order
          /l : load  checkpoint file (see below)
          /w : write checkpoint file (see below)
          /e : display end address instead of size
          /v : verbose mode

/f <filter> applies a case-insensitive search pattern.
/F <filter> works analogous, but case-sensitive.
In <filter>, the wildcards * and ? are allowed.

<operation> is one of the following:

          /p : display processes     - checkpoint: processes.dbgl
          /m : display modules       - checkpoint: modules.dbgl
          /d : display drivers       - checkpoint: drivers.dbgl
 <file> : display <file> symbols - checkpoint: symbols.dbgl

<file> is a file name, a relative path, or a fully qualified path.
Checkpoint files are loaded from and written to the current directory.
A checkpoint is an on-disk image of a DBG_LIST structure (see w2k_dbg.h).
```

В качестве дополнительной возможности w2k_sym.exe позволяет осуществлять чтение и запись в файлы контрольной точки. Контрольная точка — это просто один к одному скопированный список объектов, записанный в файл на диске. При помощи контрольных точек можно сохранять состояние системы для последующего сравнения. Файл контрольной точки содержит поле контрольной суммы CRC32, по которой содержимое файла проверяется при загрузке. Приложение w2k_sym.exe поддерживает четыре контрольные точки в текущем каталоге, по одной для четырех основных параметров программы, то есть списков процессов, модулей, драйверов и идентификаторов.

Внутренняя организация файлов идентификаторов Microsoft

Это замечательно, что Microsoft предоставляет стандартный интерфейс для доступа к файлам идентификаторов Windows 2000 независимо от их внутреннего формата. Однако бывают ситуации, в которых требуется непосредственный доступ к содержимому этих файлов для дополнительного управления их данными. В этом разделе описывается, как структурированы данные в файлах идентификаторов форматов .dbg и .pdb. Также здесь представлены библиотека DLL и клиентское

приложение-пример, при помощи которых можно искать и просматривать символьную информацию внутри этих файлов. Да, это еще одно приложение для просмотра идентификаторов, но не беспокойтесь — я не собираюсь утомлять вас незначительными переделками известного кода. Этот браузер идентификаторов совершенно отличен от предыдущего.

Декорирование идентификаторов

Файлы идентификаторов Microsoft хранят имена идентификаторов в так называемой декорированной[1] (decorated) форме, то есть к имени спереди или сзади могут быть добавлены дополнительные последовательности символов, передающие информации о типе и способе использования идентификатора. В табл. 1.4 перечислены наиболее распространенные варианты декорирования имен. Идентификаторы, генерируемые компилятором языка C, как правило, предваряются символом подчеркивания или @ в зависимости от вида соглашения о вызовах[2] (calling convention), определяющим схему передачи параметров. Символ @ означает функцию с соглашением __fastcall, символ подчеркивания — функцию с соглашением __stdcall или __cdecl. Поскольку при соглашениях __fastcall и __stdcall обязанность по очистке стека аргументов возлагается на вызываемую функцию, таким функциям присваиваются имена, включающие также число байт аргументов, помещенных в стек вызывающей функцией. Эта информация присоединяется к имени идентификатора в десятичной нотации, отделенная от имени символом @. В такой схеме глобальные переменные трактуются как функции с соглашением __cdecl, то есть их идентификаторы начинаются с символа подчеркивания, а в конце информация об аргументах не добавляется.

Таблица 1.4. Варианты декорирования идентификаторов

Пример	Описание
Symbol	Недекорированный идентификатор (может быть объявлен в модуле на ассемблере)
_symbol	функция __cdecl или глобальная переменная
_symbol@N	функция __stdcall с N байт аргументов
@symbol@N	функция __fastcall с N байт аргументов
__imp__symbol	шлюзы импорта функции __cdecl или переменной
__imp__symbol@N	шлюзы импорта функции __stdcall с N байт аргументов
__imp__@symbol@N	шлюзы импорта функции __fastcall с N байт аргументов
?symbol	идентификатор C++ со встроенной информацией о типах аргументов
__@@_PchSym_symbol	идентификатор PCH

У некоторых символьных имен есть префикс __imp или __imp_@. Такие идентификаторы присваиваются шлюзам импорта (import thunk), представляющим со-

[1] Может быть, вам этот термин не очень понравится, но я слышал его в употреблении профессионалов. Вариант «преобразование» (или какая-нибудь «модификация») мне не очень нравится: слишком широко можно его трактовать. — *Примеч. перев.*

[2] Также встречается «соглашéние о передаче парáметров». — *Примеч. перев.*

бой указатели на функции или переменные в других модулях. Шлюзы импорта способствуют динамической компоновке идентификаторов, экспортируемых другими компонентами на этапе выполнения, независимо от фактического адреса загрузки соответствующего модуля. При загрузке модуля механизм работы загрузчика обеспечивает привязку указателей шлюзов действительным адресам точек входа. Преимущество шлюзов импорта в том, что адресную привязку (fixup) каждой импортируемой функции или переменной требуется сделать только один раз для каждого идентификатора, все ссылки на этот внешний идентификатор будут направляться через его шлюз. Следует отметить, что реализация шлюзов импорта необязательна. Компилятор сам решает, нужно ли минимизировать адресные привязки при помощи реализации шлюзов или минимизировать использование памяти, экономя место, необходимое для шлюзов. Как показано в табл. 1.4, к локальным и импортируемым идентификаторам применяются одни и те же правила добавления символов (спереди и сзади), за тем исключением, что к шлюзам импорта добавляется дополнительный префикс __imp_ (с двумя лидирующими знаками подчеркивания!).

Проблемы, которые возникают при избавлении от добавленных при декорировании символов для идентификаторов в imagehlp.dll, легко можно продемонстрировать при помощи программы-примера w2k_sym.exe из предыдущего раздела, поскольку она в конечном итоге основывается на API imagehelp.dll, обращаясь к нему через библиотеку w2k_dbg.dll. Если выполнить команду w2k_sym /v/n/f __* ntoskrnl.exe, чтобы w2k_sym.exe отобразила отсортированный список имен, начинающихся с двух символов подчеркивания, вы увидите нечто похожее на список в примере 1.6. Ряд идентификаторов вверху таблицы, начинающихся с __, выглядит весьма странно. Результат выполнения команды Kernel Debugger ln 8047F798 выглядит как ntoskrnl!__, что немногим лучше. Первоначальное декорированное имя идентификатора по адресу 0x8047F798 на самом деле выглядит как ___@@_PchSym_@00@UmgUkirezgvUmglhUlyUfkUlyqUrDIGUlykOlyq@ob, поэтому, видимо, imagehlp.dll просто выбросила все символы, кроме двух из трех лидирующих символов подчеркивания.

Пример 1.6. Результаты выполнения команды w2k_sym /v/n/f __* ntoskrnl.exe

```
     #    ADDRESS             SIZE NAME
-------------------------------------------------------------
6870: 8047F798                 4 __
6871: 80480B8C                14 __
6872: 8047E724                 4 __
6873: 80471FE0                 4 __
6874: 804733B8                28 __
6875: 804721D0                20 __
6876: B04759A4                 4 __
6877: 80480004                1C __
6878: 8047DA8C                14 __
6879: 8047238C                 4 __
6880: 8047E6D4                 4 __
6881: 804755D4                 4 __
6882: 80471700                 4 __decimal_point
6883: 80471704                 4 __decimal_point_length
6884: 80471FC0                 4 __fastflag
...
```

Еще лучшим примером может служить команда w2k_sym /v/n/ _imp_* ntoskrnl.exe, выводящая все идентификаторы, начинающиеся с последовательности символов _imp_. Полученный список, частично приведенный в примере 1.7, содержит шлюзы импорта ntoskrnl.exe. Список опять начинается с длинной последовательности сомнительных имен, и опять Kernel Debugger нам не поможет, поскольку он сообщит для этих адресов точно такие же имена. Если теперь я сообщу вам, что первоначальное имя идентификатора по адресу 0x804005A4 выглядит как ___imp_@ExReleaseFastMutex@4, что вы на это скажете? Потерялся один из лидирующих символов подчеркивания и пропущена вся оставшаяся часть строки, начинающаяся с символа @. Кажется, что алгоритм обратного преобразования в модуле imagehlp.dll некорретктно обрабатывает символы @. Причина в том, что @ — это не только префикс имен функций __fastcall, но и разделитель, за которым следует часть строки с размером стека аргументов в именах функций __fastcall и __stdcall. Легко видеть, что используемый алгоритм ищет лидирующий символ подчеркивания и символ @, ошибочно полагая, что оставшаяся часть строки задает количество байт в стеке аргументов вызывающей функции. Следовательно, все длинные идентификаторы PCH урезаются до двух символов подчеркивания, а шлюзы импорта __fastcall сокращаются до _imp_. В обоих случаях первый лидирующий символ подчеркивания удаляется, а первый символ @ и все следующие за ним символы также отбрасываются.

Пример 1.7. Результат выполнения команды w2k_sym /v/n/ _imp_* ntoskrnl.exe

```
      #    ADDRESS             SIZE NAME
-------------------------------------------------------------
6861: 804005A4                 4 _imp_
6862: 80400584                 4 _imp_
6863: 80400594                 4 _imp_
6864: 80400524                 4 _imp_
6865: 8040059C                 4 _imp_
6866: 80400534                 4 _imp_
6867: 80400590                 4 _imp_
6868: 804004EC                 4 _imp_
6869: 80400554                 4 _imp_
6870: 80400598                 4 _imp_
6871: 80400520                 4 _imp_HalAllocateAdapterChannel
6872: 804004C0                 4 _imp_HalAllocateCommonBuffer
6873: 804004E8                 4 _imp_  HalAllProcessorsStarted
...
```

Посмотрев на эти два примера, вы можете потерять терпение и сказать: «Лучше-ка я сделаю это сам». Проблема в том, что внутренняя организация форматов файлов идентификаторов Microsoft очень плохо документирована, а некоторая символьная информация — особенно структура файлов в формате Program Database (PDB) — совершенно не документирована. В Microsoft Knowledge Base есть даже статья, недвусмысленно утверждающая:

«Формат Program Database File Format, известный также как формат файлов PDB, не документирован. Эта информация находится в собственности Microsoft».

Звучит так, что все попытки создать свою программу разбора информации об идентификаторах должны провалиться. Несмотря на это, в книге вы не найдете слов «... к сожалению, больше ничего не могу вам сказать, поскольку внутренняя структур файлов PDB мне неизвестна». Разумеется, я расскажу, как организованы файлы PDB. Однако сначала необходимо исследовать внутреннее устройство файлов .dbg, поскольку все начинается именно с них.

Внутренняя структура файлов .dbg

Символьная информация компонентов Windows NT 4.0 упакована в файлы с расширением .dbg. Имена файлов и содержащих их подкаталогов можно немедленно получить из имени файла компонента. Если корневой каталог идентификаторов системы определен как d:\winnt\symbols, полный путь к файлу идентификаторов filename.ext будет такой: d:\winnt\symbols\ext\filename.dbg. Например, идентификаторы ядра будут находиться в файле d:\winnt\symbols\exe\ntoskrnl.dbg. В Windows 2000 файлы .dbg также присутствуют, однако вся символьная информация была перенесена в отдельные файлы .pdb. Поэтому каждому компоненту Windows 2000 в корневом каталоге идентификаторов помимо файла ext\filename.dbg сопоставлен дополнительный файл ext\filename.pdb. Кроме этого отличия, содержимое файлов .dbg в системах Windows NT 4.0 и 2000 очень похоже.

К счастью, внутренняя организация файлов .dbg хотя бы частично документирована. В заголовочном файле winnt.h из комплекта Win32 Platform SDK содержатся важные определения констант и типов данных ключевых частей формата, а в библиотеке Microsoft Development Network (MSDN) есть несколько очень ценных статей об этом формате файлов. Наиболее подробно материал изложен, безусловно, в выпуске колонки «Under the Hood» Матта Питрека (Matt Pietrek) в журнале «Microsoft Systems Journal» (MSJ) за март 1999 г., переименованной в MSDN в статью «MSDN Magazine». Файл .dbg состоит главным образом из заголовка и раздела данных, размер которых может меняться. Оба раздела делятся далее на подразделы. Часть заголовка состоит из четырех основных подразделов:

1. Структура IMAGE_SEPARATE_DEBUG_HEADER, начинающаяся с двухбуквенной сигнатуры «DI» (верхняя часть листинга 1.13).
2. Массив структур IMAGE_SECTION_HEADER, по одной для каждого раздела файла PE компонента (средняя часть листинга 1.13). Количество элементов задается членом NumberOfSections структуры IMAGE_SEPARATE_DEBUG_HEADER.
3. Ряд завершающихся нулем 8-битных строк ANSI, в которых содержатся все экспортируемые идентификаторы в недекорированной форме. Размер этого подраздела задается членом ExportedNameSize структуры IMAGE_SEPARATE_DEBUG_ HEADER. Если модуль не экспортирует никаких идентификаторов, ExportedNameSize равен нулю и этот подраздел отсутствует.
4. Массив структур IMAGE_DEBUG_DIRECTORY, описывающий расположение и форматы последующих данных в файле (нижняя часть листинга 1.13). Размер этого подраздела задается членом DebugDirectorySize структуры IMAGE_SEPARATE_DEBUG_HEADER.

Листинг 1.13. Структуры заголовка файла .dbg

```
#define IMAGE_SEPARATE_DEBUG_SIGNATURE 0x4944 // "DI"
typedef struct _IMAGE_SEPARATE_DEBUG_HEADER
      {
      WORD      Signature;
      WORD      Flags;
      WORD      Machine;
      WORD      Characteristics;
      DWORD     TimeDateStamp;
      DWORD     CheckSum;
      DWORD     ImageBase;
      DWORD     SizeOfImage;
      DWORD     NumberOfSections;
      DWORD     ExportedNamesSize;
      DWORD     DebugDirectorySize;
      DWORD     SectionAlignment;
      DWORD     Reserved[2];
      }
         IMAGE_SEPARATE_DEBUG_HEADER,
      *PIMAGE_SEPARATE_DEBUG_HEADER;

//  ------------------------------------------------------

#define  IMAGE_SIZEOF_SHORT_NAME                  8

typedef struct _IMAGE_SECTION_HEADER
      {
      BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
      union
           {
           DWORD PhysicalAddress;
           DWORD VirtualSize;
           } Misc;
      DWORD     VirtualAddress;
      DWORD     SizeOfRawData;
      DWORD     PointerToRawData;
      DWORD     PointerToRelocations;
      DWORD     PointerToLinenumbers;
      WORD      NumberOfRelocations;
      WORD      NumberOfLinenumbers;
      DWORD     Characteristics;
      }
      IMAGE_SECTION_HEADER,  *PIMAGE_SECTION_HEADER;

//  ------------------------------------------------------

#define  IMAGE_DEBUG_TYPE_UNKNOWN             0
#define  IMAGE_DEBUG_TYPE_COFF                1
#define  IMAGE_DEBUG_TYPE_CODEVIEW            2
#define  IMAGE_DEBUG_TYPE_FPO                 3
#define  IMAGE_DEBUG_TYPE_MISC                4
#define  IMAGE_DEBUG_TYPE_EXCEPTION           5
#define  IMAGE_DEBUG_TYPE_FIXUP               6
#define  IMAGE_DEBUG_TYPE_OMAP_TO_SRC         7
#define  IMAGE_DEBUG_TYPE_OMAP_FROM_SRC       8
#define  IMAGE_DEBUG_TYPE_BORLAND             9
```

продолжение ⇩

Листинг 1.13 (*продолжение*)

```
#define  IMAGE_DEBUG_TYPE_RESERVED10          10
#define  IMAGE_DEBUG_TYPE_CLSID               11

typedef struct _IMAGE_DEBUG_DIRECTORY
      {
      DWORD     Characteristics;
      DWORD     TimeDateStamp;
      WORD      MajorVersion;
      WORD      MinorVersion;
      DWORD     Type;
      DWORD     SizeOfData;
      DWORD     AddressOfRawData;
      DWORD     PointerToRawData;
      }
      IMAGE_DEBUG_DIRECTORY,  *PIMAGE_DEBUG_DIRECTORY;
```

Поскольку размер подразделов не фиксируется, их абсолютные позиции в файле .dbg необходимо отсчитывать относительно размеров предыдущих подразделов. Синтаксический анализатор файла .dbg, как правило, работает по следующему алгоритму:

- структура IMAGE_SEPARATE_DEBUG_HEADEDR всегда расположена в начале файла;
- первая структура IMAGE_SECTION_HEADEDR немедленно следует за IMAGE_SEPARATE_DEBUG_HEADEDR, поэтому она всегда расположена по смещению 0x30 от начала файла;
- смещение первого экспортируемого имени вычисляется путем умножения размера структуры IMAGE_SECTION_HEADEDR на количество разделов и суммирования этого числа со смещением первого раздела заголовка. Таким образом, первая строка расположена по смещению 0x30 + (NumberOfSections * 0x28);
- смещения остальных структур данных в файле .dbg определены в элементах IMAGE_DEBUG_DIRECTORY. Смещения и размеры блоков данных задаются соответственно членами PointerToRawData и SizeOfData;

Определения IMAGE_DEBUG_TYPE_* из листинга 1.13 отражают разнообразные форматы данных, которые могут находиться в файле .dbg. Однако файлы идентификаторов Windows NT 4.0, как правило, содержат данные только четырех из этих типов: IMAGE_DEBUG_TYPE_COFF, IMAGE_DEBUG_TYPE_CODEVIEW, IMAGE_DEBUG_TYPE_FPO и IMAGE_DEBUG_TYPE_MISC. В файлах .dbg Windows 2000 к этим типам, как правило, добавляются IMAGE_DEBUG_TYPE_OMAP_TO_SRC, IMAGE_DEBUG_TYPE_OMAP_FROM_SRC и недокументированный тип с идентификатором 0x100. Если вас интересует только сопоставление идентификаторов адресам или просмотр идентификаторов, потребуются только элементы каталога IMAGE_DEBUG_TYPE_CODEVIEW, IMAGE_DEBUG_TYPE_OMAP_TO_SRC и IMAGE_DEBUG_TYPE_OMAP_FROM_SRC.

На прилагающемся к книге компакт-диске находится библиотека-пример w2k_img.dll, осуществляющая разбор файлов .dbg и .pdb и экспортирующая довольно интересные функции для разработчиков инструментов отладки. Исходный код этой DLL можно найти в каталоге \src\w2k_img компакт-диска. Важная особенность w2k_img.dll: она разработана для *всех* платформ Win32. Сюда относятся не только Windows 2000 и Windows NT 4.0, но также и Windows 95 и 98. Как и все добропорядочные программы в мире Win32, DLL предоставляет раздельные точки входа для строк ANSI и Unicode. По умолчанию клиентское приложение использует

функции ANSI. Если в исходный код приложения включить строку #define UNICODE, прозрачно будут выбраны точки входа Unicode. Клиентские приложения для общей платформы Win32 должны пользоваться только ANSI. Приложения только для Windows 2000/NT могут перейти на Unicode для лучшей производительности.

На компакт-диске с примерами содержится, кроме того, приложение SBS Windows 2000 CodeView Decompiler, файлы проекта Microsoft Visual C/C++ находятся в каталоге \src\w2k_cv. Это очень простое приложение, раскрывающее файлы .dbg и .pdb и выводящее их содержимое в окно консоли. При чтении этого раздела оно поможет ознакомиться с реальными примерами обсуждаемых структур данных. Приложение w2k_cv.exe постоянно обращается к нескольким функциям API из модуля w2k_img.dll.

В листинге 1.14 показана одна из основных структур данных, определенных в w2k_img.h. Структура IMG_DBG фактически состоит из первых двух разделов заголовка файла .dbg, то есть основного заголовка фиксированного размера и массива заголовков раздела PE. Макрос IMG_DBG__() вычисляет действительный размер этой структуры по заданному количеству разделов. Полученное значение задает смещение в файле подраздела экспортируемых имен.

Несколько функций API из библиотеки w2k_img.dll ожидают указатель на инициализированную структуру IMG_DBG. Функция imgDbgLoad() (здесь она не приведена) выделяет память и корректно инициализирует структуру IMG_DBG данными переданного ей файла .dbg, передавая обратно указатель на эту структуру[1]. imgDbgLoad() очень тщательно проверяет данные, чтобы убедиться в полноте и корректности файла. Инициализированная структура IMG_DBG может быть передана нескольким функциям синтаксического разбора, возвращающим линейные адреса наиболее часто используемых компонентов файла .dbg. Например, функция imgDbgExports() из листинга 1.15 вычисляет линейный адрес последовательности экспортируемых имен, расположенных сразу за массивом структур IMAGE_SECTION_HEADER. Она также подсчитывает число имен, просматривая последовательность строк до конца раздела и (необязательно) записывает это значение в переменную, на которую указывает аргумент pdCount.

Листинг 1.14. Структура IMG_DBG и связанный с ней макрос

```
typedef struct _IMG_DBG
      {
      IMAGE_SEPARATE_DEBUG_HEADER  Header;
      IMAGE_SECTION_HEADER         aSections [];
      }
      IMG_DBG, *PIMG_DBG, **PPIMG_DBG;

#define IMG_DBG_ sizeof (IMG_DBG)
#define IMG_DBG__(_n) \
          (IMG_DBG_ + ((_n) * IMAGE_SECTION_HEADER_))

#define IMG_DBG_DATA(_p,_d) \
          ((PVOID) ((PBYTE) (_p) + (_d)->PointerToRawData))
```

[1] Так в прототипе функции (возвращает PVOID). Слова «returns structure» вообще-то можно понять и как возврат переменной типа struct. Поскольку автор, скорее всего, везде передает структуры через указатели (что естественно в системном ПО), «возвращая структуру» не вызывало бы путаницы. В том числе так перевожу и в дальнейшем, кроме того, автор обычно приводит коды. Здесь же в русском контексте удобнее написать так. — *Примеч. перев.*

Листинг 1.15. Функция API imgDbgExports()

```
PBYTE WINAPI imgDbgExports (PIMG_DBG  pid,
                            PDWORD    pdCount)
      {
      DWORD i, j;
      DWORD dCount       = 0;
      PBYTE pbExports    = NULL;

      if (pid != NULL)
           {
           pbExports = (PBYTE) pid->aSections
                     + (pid->Header.NumberOfSections
                          * IMAGE_SECTION_HEADER_);

           for (i = 0;
                i < pid->Header.ExportedNamesSize; i = j)
               {
               if (!pbExports [j = i]) break;

               while ((j < pid->Header.ExportedNamesSize) &&
                         pbExports [j++]);

               if ((j > i) && (!pbExports [j-1])) dCount++;
               }
           }
      if (pdCount != NULL) *pdCount = dCount;
      return pbExports;
      }
```

В листинге 1.16 определены еще две функции API, определяющие расположение элементов каталога по их идентификаторам IMAGE_DEBUG_TYPE_*. Функция imgDbgDirectories() возвращает базовый адрес массива IMAGE_DEBUG_DIRECTORY, в то время как imgDbgdirectory возвращает указатель на первый элемент каталога с заданным идентификатором типа или NULL, если таковых элементов не оказалось.

Листинг 1.16. Функции API imgDbgDirectories() и imgDbgDirectory()

```
PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectories (PIMG_DBG  pid,
                                                 PDWORD    pdCount)
      {
      DWORD                  dCount = 0;
      PIMAGE_DEBUG_DIRECTORY pidd   = NULL;

      if (pid != NULL)
           {
           pidd = (PIMAGE_DEBUG_DIRECTORY)
                    ((PBYTE) pid
                     + IMG_DBG__ (pid->Header.NumberOfSections)
                     + pid->Header.ExportedNamesSize);

           dCount = pid->Header.DebugDirectorySize
                        / IMAGE_DEBUG_DIRECTORY_;
           }
      if (pdCount != NULL) *pdCount = dCount;
      return pidd;
      }
```

```
//  ------------------------------------------------------

PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectory (PIMG_DBG   pid,
                                               DWORD      dType)
      {
      DWORD                   dCount, i;
      PIMAGE_DEBUG_DIRECTORY  pidd = NULL;

      if ((pidd = imgDbgDirectories (pid, &dCount)) != NULL)
           {
           for (i = 0; i < dCount; i++, pidd++)
                {
                if (pidd->Type == dType) break;
                }
           if (i == dCount) pidd = NULL;
           }
      return pidd;
      }
```

При помощи функции imgDbgDirectory() можно осуществлять поиск данных CodeView в файле .dbg. Это осуществляется функцией imgDbgCv() из листинга 1.17. Она вызывает функцию imgDbgDirectory(), передавая ей тип идентификатора IMAGE_DEBUG_TYPE_CODEVIEW, и преобразует смещение данных элемента IMAGE_DEBUG_DIRECTORY в абсолютный линейный адрес при помощи макроса IMG_DBG_DATA(), показанного в листинге 1.14. Этот макрос добавляет смещение к базовому адресу структуры IMG_DBG и приводит ее тип к указателю типа PVOID. imgDbgCv() копирует размер подсекции CodeView в *pdSize, если переданный аргумент pdSize не равен NULL. Внутренняя организация данных CodeView описана ниже.

Функции API для других подразделов данных выглядят весьма похоже. В листинге 1.18 показаны функции imgDbgOmapToSrc() и imgDbgOmapFromSrc() и структуры, с которыми они оперируют — OMAP_TO_SRC и OMAP_FROM_SRC. Позже нам эти структуры понадобятся для вычисления линейного адреса идентификатора из данных CodeView. Поскольку данные OMAP представляют собой массив структур постоянной длины, обе функции API возвращают не просто размер подраздела, а число его элементов, деля общий размер подраздела на размер одного элемента. Полученное значение копируется в *pdCount, если аргумент pdCount не равен NULL.

Листинг 1.17. Функция API imgDbgCv()

```
PCV_DATA WINAPI imgDbgCv (PIMG_DBG   pid,
                          PDWORD     pdSize)
      {
      PIMAGE_DEBUG_DIRECTORY  pidd;
      DWORD                   dSize = 0;
      PCV_DATA                pcd   = NULL;

      if ((pidd = imgDbgDirectory (pid,
                  IMAGE_DEBUG_TYPE_CODEVIEW))
           != NULL)
           {
           pcd   = IMG_DBG_DATA (pid, pidd);
           dSize = pidd->SizeOfData;
           }
```

продолжение

Листинг 1.17 (*продолжение*)

```
    if (pdSize != NULL) *pdSize = dSize;
    return pcd;
    }
```

Листинг 1.18. Функции API imgDbgOmapToSrc() и imgDbgOmapFromSrc()

```
typedef struct _OMAP_TO_SRC
    {
    DWORD dTarget;
    DWORD dSource;
    }
    OMAP_TO_SRC, *POMAP_TO_SRC, **PPOMAP_TO_SRC;

#define OMAP_TO_SRC_ sizeof (OMAP_TO_SRC)

//  ------------------------------------------------------

typedef struct _OMAP_FROM_SRC
    {
    DWORD dSource;
    DWORD dTarget;
    }
    OMAP_FROM_SRC, *POMAP_FROM_SRC, **PPOMAP_FROM_SRC;

#define OMAP_FROM_SRC_ sizeof (OMAP_FROM_SRC)

//  ------------------------------------------------------

POMAP_TO_SRC WINAPI imgDbgOmapToSrc (PIMG_DBG   pid,
                                     PDWORD     pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY  pidd;
    DWORD                   dCount = 0;
    POMAP_TO_SRC            pots   = NULL;

    if ((pidd = imgDbgDirectory (pid,
                                 IMAGE_DEBUG_TYPE_OMAP_TO_SRC))
        != NULL)
        {
        pots   = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_TO_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pots;
    }

//  ------------------------------------------------------

POMAP_FROM_SRC WINAPI imgDbgOmapFromSrc (PIMG_DBG   pid,
                                         PDWORD     pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY  pidd;
    DWORD                   dCount = 0;
    POMAP_FROM_SRC          pofs   = NULL;

    if ((pidd = imgDbgDirectory (pid,
                         IMAGE_DEBUG_TYPE_OMAP_FROM_SRC))
```

```
        != NULL)
        {
        pofs    = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_FROM_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pofs;
    }
```

Подразделы CodeView

CodeView — собственный формат Microsoft для отладочной информации. За время нескольких лет развития компилятора и компоновщика Microsoft C/C++ он подвергался неоднократной переработке. Внутренняя структура различных версий CodeView кардинально различается. Тем не менее во всех версиях CodeView в начале данных присутствует 32-битная сигнатура, однозначно определяющая этот формат данных. Файлы идентификаторов Windows NT 4.0 используют формат NB09, соответствующий CodeView 4.10. Файлы Windows 2000 содержат данные CodeView NB10, которые, как я покажу далее, просто ссылаются на отдельный файл .pdb.

Данные CodeView в формате NB09 делятся на каталог и элементы младшего уровня. Как отмечает Матт Питрек в своей статье в *MSJ* о файлах .dbg, большинство основных структур CodeView определены в наборе заголовочных файлов-примеров, входящих в состав комплекта Platform SDK. Если эти примеры из SDK у вас установлены, в каталоге \Program Files\Microsoft Platform SDK\Samples\SdkTools\Image\Include вы найдете немало интересных файлов. Для синтаксического разбора CodeView потребуются файлы cvexefmt.h и cvinfo.h. К сожалению, судя по дате 09.07.1994, эти файлы уже достаточно долго не обновлялись. Довольно удивительно, что все имена определенных в файле cvexefmt.h структур начинаются с букв OMF, сокращение от Object Module Format. OMF — стандартный формат файлов, используемый 16-разрядными файлами DOS и Windows в форматах .obj и .lib. Начиная с версий средств разработки Microsoft для Win32, этот формат был заменен на Common Object File Format (COFF, подробнее см. Gircys, 1988).

Хотя первоначальный формат OMF сегодня уже устарел, необходимо отдать ему должное: это был хорошо продуманный формат файлов. Одной из целей при его проектировании была максимальная экономия памяти и дискового пространства. Формат обладает еще одним важным свойством — его могут успешно читать приложения, даже если им полностью не известна организация всех частей файла. Базовая структура данных формата OMF — это предваряемая тегом запись, начинающаяся с байта тега, определяющего тип содержащихся в записи данных, и 16-битного слова длины, содержащего количество последующих байт. Такая схема позволяла программе чтения формата OMF переходить от записи к записи, отбирая только записи с нужными типами данных. Эту же парадигму Microsoft приняла и для своего формата CodeView, что объясняет наличие префикса OMF у имен структур CodeView в файле cvexefmt.h. Хотя записи формата CodeView имеют весьма мало общего с первоначальными записями OMF, сохранилось главное свойство OMF, дающее возможность чтения формата без точного знания структуры всего его содержимого.

В листинге 1.19 приведены определения различных основных структур формата CodeView, взятые из файла w2k_img.h. Некоторые структуры слабо соотносятся со структурами файлов cvexefmt.h и cvinfo.h, так как они приспособлены для нужд функций API из библиотеки w2k_img.dll. Структура CV_HEADER присутствует во всех данных формата CodeView независимо от версии формата. 32-битный член этой структуры Signature определяет версию формата, такую как CV_SIGNATURE_NB09 или CV_SIGNATURE_NB10. Член lOffset задает смещение каталога CodeView относительно адреса заголовка. В файлах идентификаторов Windows NT 4.0 в формате NB09 это значение, по-видимому, все время равно восьми, показывая, что каталог расположен сразу же после заголовка. Файлы идентификаторов Windows 2000 содержат данные в формате NB10, смещение lOffset равно нулю. Этот формат данных будет подробно обсужден позже (в этой главе).

Листинг 1.19. Структуры данных формата CodeView

```
#define CV_SIGNATURE_NB   'BN'
#define CV_SIGNATURE_NB09 '90BN'
#define CV_SIGNATURE_NB10 '01BN'

//  ------------------------------------------------------

typedef union _CV_SIGNATURE
      {
      WORD wMagic:        // 'BN'
      DWORD     Version;  // 'xxBN'
      BYTE abText [4];    // "NBxx"
      }
      CV_SIGNATURE, *PCV_SIGNATURE, **PPCV_SIGNATURE;

#define CV_SIGNATURE_ sizeof (CV_SIGNATURE)

//  ------------------------------------------------------

typedef struct _CV_HEADER
      {
      CV_SIGNATURE  Signature;
      LONG          lOffset;
      }
      CV_HEADER, *PCV_HEADER, **PPCV_HEADER;

#define CV_HEADER_ sizeof (CV_HEADER)

//  ------------------------------------------------------

typedef struct _CV_DIRECTORY
      {
      WORD  wSize:        // в байтах, включая этот член
      WORD  wEntrySize;   // в байтах
      DWORD dEntries;
      LONG  lOffset;
      DWORD dFlags;
      }
      CV_DIRECTORY, *PCV_DIRECTORY, **PPCV_DIRECTORY;

#define CV_DIRECTORY_ sizeof (CV_DIRECTORY)
```

```
//  ------------------------------------------------------

#define sstModule          0x0120 // CV_MODULE
#define sstGlobalPub       0x012A // CV_PUBSYM
#define sstSegMap          0x012D // SV_SEGMAP

//  ------------------------------------------------------

typedef struct _CV_ENTRY
      {
      WORD  wSubSectionType;       // sst*
      WORD  wModuleIndex;          // -1, если не применим
      LONG  lSubSectionOffset;     // относительно CV_HEADER
      DWORD dSubSectionSize;       // в байтах, без учета
                                   // байтов заполнения
      }
      CV_ENTRY, *PCV_ENTRY, **PPCV_ENTRY;

#define CV_ENTRY_ sizeof (CV_ENTRY)

//  ------------------------------------------------------

typedef struct _CV_NB09 // CodeView 4.10
      {
      CV_HEADER      Header;
      CV_DIRECTORY   Directory;
      CV_ENTRY       Entries [];
      }
      CV_NB09, *PCV_NB09, **PPCV_NB09;

#define CV_NB09_ sizeof (CV_NB09)
```

Каталог CodeView NB09 состоит из единственной структуры CV_DIRECTORY, за которой следует массив, состоящий из структур CV_ENTRY. Это отражено в структуре CB_NB09, определенной в конце листинга 1.19, содержащей заголовок CodeView, каталог и массив Entries[], размер которого определяется членом dEntries структуры CV_DIRECTORY. Каждый элемент CV_ENTRY соответствует подразделу CodeView, тип которого хранится в члене wSubSectionType. В файле cvexefmt.h определено не менее 21 типа подразделов. Однако файлы идентификаторов Windows NT 4.0 используют только три из них: sstModule (0x0120), sstGlobalPub (0x012A) и sstSegMap (0x12D). Как правило, в файле идентификаторов присутствует несколько подразделов sstModule и только по одному подразделу sstGlobalPub и sstSegMap. Как можно догадаться из имени, в подразделе sstGlobalPub хранятся глобальные открытые идентификаторы соответствующего модуля.

Функция API imgCvEntry() (листинг 1.20) из модуля w2k_img.dll позволяет легко находить записи каталога CodeView с требуемым типом. Ее аргумент pc09 указывает на структуру CV_NB09, то есть на сигнатуру формата NB09 блока данных CodeView внутри файла .dbg. Аргумент dType определяет один из идентификаторов типов подразделов CodeView sst*, а значение dIndex определяет конкретный подраздел в случае нескольких подразделов одного типа. Поэтому ненулевое значение dIndex имеет смысл только тогда, когда dType равен setModule.

Листинг 1.20. Функции API imgCvEntry() и imgCvSymbois()

```
PCV_ENTRY WINAPI imgCvEntry (PCV_NB09    pc09,
                             DWORD       dType,
                             DWORD       dIndex)
      {
      DWORD         i, j;
      PCV_ENTRY pce = NULL;

      if ((pc09 != NULL) &&
          (pc09->Header.Signature.dVersion  ==
           CV_SIGNATURE_NB09))
          {
          for (i = j = 0; i < pc09->Directory.dEntries; i++)
              {
              if ((pc09->Entries [i].wSubSectionType == dType)
                  &&
                  (j++ == dIndex))
                  {
                  pce = pc09->Entries + i;
                  break;
                  }
              }
          }
      return pce;
      }

//  ------------------------------------------------------

PCV_PUBSYM WINAPI  imgCvSymbols (PCV_NB09  pc09,
                                 PDWORD    pdCount,
                                 PDWORD    pdSize)
      {
      PCV_ENTRY      pce;
      PCV_PUBSYM     pcp1;
      DWORD          i;
      DWORD          dCount    = 0;
      DWORD          dSize     = 0;
      PCV_PUBSYM     pcp       = NULL;

      if ((pce = imgCvEntry (pc09, sstGlobalPub, 0))
          != NULL)
          {
          pcp = CV_PUBSYM_OATA ((PBYTE) pc09
                                          + pce->lSubSectionOffset);

          dSize = pce->dSubSectionSize;

          for (i  = 0; dSize - i >= CV_PUBSYM_;
               i += CV_PUBSYM_SIZE (pcp1))
              {
              pcp1 = (PCV_PUBSYM) ((PBYTE) pcp + i);
              if (dSize - i < CV_PUBSYM_SIZE (pcp1)) break;
              if (pcp1->Header.wRecordType == CV_PUB32)
                  dCount++;
              }
          }
      if (pdCount != NULL) *pdCount = dCount;
      if (pdSize  != NULL) *pdSize  = dSize;
      return pcp;
      }
```

Идентификаторы CodeView

В нижней половине листинга 1.20 показана функция imgCvSymbols(), возвращающая указатель на первую символьную запись CodeView. Подраздел sstGlobalPub состоит из заголовка фиксированной длины CV_SYMHASH, за которым следует ряд записей переменной длины CV_PUBSYM. Определения обоих типов включены в листинг 1.21. Сначала imgCvSymbols() вызывает imgCvEntry(), чтобы найти запись CV_ENTRY, член wSubSectionType которой равен sstGlobalPub. Если такая найдется, для пропуска находящейся вначале структуры CV_SYMHASH применяется макрос CV_PUBSYM_DATA() из листинга 1.4. После этого imgCvSymbol() подсчитывает количество идентификаторов, проходя по списку записей CV_PUBSYM, вычисляя размер каждой записи при помощи макроса CV_PUBSYM_SIZE() из листинга 1.21.

Последовательность записей CV_PUBSYM немного напоминает содержимое объектного файла в формате OMF. Как уже отмечалось, поток данных OMF состоит из записей переменной длины, начинающихся с байта тега и 16-битного слова со значением длины записи. Записи CV_PUBSYM организованы аналогично. Они начинаются с заголовка OMF_HEADER, содержащего члены wRecordSize и wRecordType. По-сути, это разновидность принципа организации OMF, отличие только в том, что слово длины расположено первым, а байт тега был расширен до 16 бит. Последнюю часть структуры CV_PUBSYM занимает имя идентификатора в формате PASCAL, как обычно определяется строка в записи OMF. Строка PASCAL состоит из лидирующего байта длины, за которым следует от 0 до 255 8-битных символов. В отличие от строк C завершающий нуль не добавляется. Запись CV_PUBSYM заканчивается после последнего символа Name, однако дополняется байтами до очередной 32-битной границы. Эти байты учтены в значении wRecordSize в заголовке записи OMF_HEADER. Обратите внимание на то, что wRecordSize хранит размер записи CV_PUBSYM, *за исключением* самого члена wRecordSize. Вот почему макрос CV_PUBSYM_SIZE() в листинге 1.21 добавляет к значению wRecordSize величину sizeof (WORD), получая итоговый размер записи.

Листинг 1.21. Структуры CV_SYMHASH и CV_PUBSYM

```
typedef struct _CV_SYMHASH
      {
      WORD  wSymbolHashIndex;
      WORD  wAddressHashIndex;
      DWORD dSymbolInfoSize;
      DWORD dSymbolHashSize;
      DWORD dAddressHashSize;
      }
      CV_SYMHASH, *PCV_SYMHASH, **PPCV_SYMHASH;

#define CV_SYMHASH_ sizeof (CV_SYMHASH)
//  ----------------------------------------------------

typedef struct _OMF_HEADER
      {
      WORD wRecordSize; // в байтах, без учета этого члена
      WORD wRecordType;
      }
      OMF_HEADER, *POMF_HEADER, **PPOMF_HEADER;
```

продолжение

Листинг 1.21 (*продолжение*)

```
#define OMF_HEADER_ sizeof (OMF_HEADER)

//  ------------------------------------------------------

typedef struct _OMF_NAME
      {
      BYTE bLength;       .// в байтах, без учета этого члена
      BYTE abName [];
      }
      OMF_NAME, *POMF_NAME, **PPOMF_NAME;

#define OMF_NAME_ sizeof (OMF_NAME)
//  ------------------------------------------------------

#define S_PUB32  0x0203
#define S_ALIGN  0x0402

#define CV_PUB32 S_PUB32

//  ------------------------------------------------------

typedef struct _CV_PUBSYM
      {
      OMF_HEADER     Header;
      DWORD          dOffset;
      WORD           wSegment:      // индекс подраздела,
                                    // начинающийся с 1
      WORD           wTypeIndex:    // 0
      OMF_NAME       Name:          // заполнение нулями до
                                    // следующего DWORD
      }
      CV_PUBSYM, *PCV_PUBSYM, **PPCV_PUBSYM;

#define CV_PUBSYM_ sizeof (CV_PUBSYM)

#define CV_PUBSYM_DATA(_p) \
          ((PCV_PUBSYM) ((PBYTE) (_p) + CV_SYMHASH_))

#define CV_PUBSYM_SIZE(_p) \
          ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define CV_PUBSYM_NEXT(_p) \
          ((PCV_PUBSYM) ((PBYTE) (_p) + CV_PUBSYM_SIZE (_p)))
```

При просмотре потока записей CV_PUBSYM вам, как правило, будут встречаться два типа записей: S_PUB32 (0x0203) или S_ALIGN (0x0402). На последний тип можно смело не обращать внимания, поскольку это только байты заполнения, а сама символьная информация содержится в записях S_PUB32. Помимо идентификатора Name интерес представляют члены wSegment и dOffset. wSegment задает начинающийся с единицы индекс, определяющий раздел файла PE, содержащий идентификатор. Если из этого индекса вычесть единицу, полученное значение можно использовать в качестве индекса в массиве IMAGE_SECTION_HEADERS в начале файла .dbg. dOffset содержит адрес идентификатора относительно начала его раздела PE.

В данном контексте адрес идентификатора — это точка входа функции или базовый адрес глобальной переменной. Как правило, значение dOffset можно просто добавить к значению VirtualAddress соответствующего заголовка IMAGE_SECTION_HEADER, что дает адрес идентификатора относительно базового адреса модуля. Однако если файл .dbg содержит подразделы IMAGE_DEBUG_TYPE_OMAP_TO_SRC и IMAGE_DEBUG_TYPE_OMAP_FROM_SRC, значение dOffset должно пройти еще через один уровень преобразования. Работа с таблицами OMAP будет рассмотрена позже, после описания формата файлов PDB.

Идентификаторы в подразделе sstGlobalPub выглядят как расположенные случайно. Принцип их расположения мне неизвестен. Я могу лишь с уверенностью сказать, что идентификаторы не отсортированы по номеру раздела, смещению или имени. Не следует строить догадки по поводу упорядоченности идентификаторов. Если вашему приложению требуется определенный порядок их расположения, необходимо отсортировать записи с идентификаторами самостоятельно. Библиотека-пример w2k_img.dll, находящаяся на прилагающемся к книге компакт-диске, по умолчанию обеспечивает сортировку идентификаторов по трем различным критериям: по адресу, по имени с учетом регистра и по имени без учета регистра.

Внутренняя структура файлов .pdb

После установки файлов идентификаторов Windows 2000 первое, что бросается в глаза, это то, что с каждым модулем теперь связано два файла: один, как обычно, с расширением .dbg, а также дополнительный файл с раширением .pdb. Если заглянуть внутрь одного из файлов .pdb, в самом начале будет расположена строка «Microsoft C/C++ program database 2.0». То есть PDB — это сокращение от Program Database (база данных программы). Поиск дополнительной информации о структуре формата PDB в библиотеке MSDN или в Интернете не дал особых результатов, за исключением статьи из «Microsoft Knowledge Base», относящей этот формат к собственности Microsoft (Microsoft Corporation, 2000d). Даже гуру в области Windows Матт Питрек соглашается с этим:

«Формат таблиц идентификаторов PDB открыто не документирован. (Даже я не знаю точную структуру формата, особенно если учесть, что он продолжает развиваться с каждым новым выпуском Visual C++.)».

Ну, формат PDB может развиваться с каждым выпуском visual C/C++, но для текущей версии Windows 2000 я точно могу вам сказать, какова структура файлов идентификаторов PDB. Наверное, формат PDB здесь описывается впервые. Но сначала позвольте мне объяснить, как связаны друг с другом файлы .dbg и .pdb.

Файлы .dbg в Windows 2000 обладают одним замечательным свойством: они содержат очень крохотный, практически не существенный подраздел CodeView. В примере 1.8 показаны все данные из файла ntoskrnl.dbg, полученные при помощи утилиты w2k_dump.exe, расположенной в каталоге \src\w2k_dump компакт-диска с примерами. Это все — всего лишь 32 байта. Как обычно, подраздел начинается со структуры CV_HEADER, содержащей сигнатуру версии формата CodeView, на

этот раз это NB10. Библиотека MSDN (Microsoft 2000a) не очень-то много сообщает нам об этой версии:

«NB10 сигнатура исполняемого файла, отладочная информация которого хранится в отдельном файле PDB. Согласован с форматами, определнными в NB09 или NB11». (MSDN Library — April 2000\Specifications\Technologies and Languages\Visual C++ 5.0 Symbolic Debug Information Specification\Debug Information Format)

Я не знаком со структурой формата NB11, однако формат PDB не имеет почти ничего общего с обсужденным выше форматом NB09! Из первого предложения ясно, почему блок данных NB10 настолько мал. Вся соответствующая информация перенесена в отдельный файл, поэтому основное назначение раздела CodeView — предоставить ссылку на собственно данные. В примере 1.8 символьную информацию следует искать в формате ntoskrnl.pdb в каталоге установленных идентификаторов Windows 2000.

Пример 1.8. Шестнадцатеричный снимок подраздела CodeView формата PDB

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 01234567B9ABCDEF
---------|-------------------------:-------------------------|-----------------
00006590 | 4E 42 31 30-00 00 00 00 : 20 7D 23 38-54 00 00 00 | NB10....  }#8T...
000065A0 | 6E 74 6F 73-6B 72 6E 6C : 2E 70 64 62-00 00 00 00 | ntoskrnl.pdb....
```

Если вас интересует, для чего служат остальные данные в примере 1.8, листинг 1.22 удовлетворит ваше любопытство. Заголовок CV_HEADER не нуждается в объяснениях. Следующие два члена по смещениям 0x8 и 0xC называются dSignature и dAge и играют большую роль при компоновке файлов .dbg и .pdb. dSignature представляет из себя 32-битную временную метку в стиле UNIX, определяющую время и дату сборки отладочной информации в секундах, прошедших с 01.01.1970. Функции imgTimeUnpack() и imgTimePack() из библиотеки w2k_img.dll преобразовывают этот необычный для Windows формат даты/времени в стандартный формат и обратно. Назначение члена dAge мне не до конца ясно. По-видимому, его значение сначала равно единице и увеличивается на 1 при каждой перезаписи данных PDB. Взятые вместе значения dSignature и dAge образуют 64-битный идентификатор, на основе значения которого отладчики могут проверять соответствие данного файла PDB файлу .dbg, который на него ссылается. Файл PDB содержит копии обоих значений в одном из своих потоков данных, поэтому отладчик может отказаться обрабатывать пару файлов .dbg/.pdb, если значения dSignature и dAge у них не совпадают.

При встрече с неизвестным форматом данных первое, что нужно сделать, это пропустить несколько образцов данных этого формата через средства просмотра данных в виде шестнадцатеричного снимка. Для этого очень удобно воспользоваться утилитой w2k_dump.exe, расположенной на прилагающемся к книге компакт-диске. При изучении такого снимка файла Windows 2000 в формате PDB выявляются его интересные свойства:

- файл, по-видимому, разделен на блоки фиксированного размера — как правило, по 0x400 байт;
- некоторые блоки состоят из длительных последовательностей единичных бит, время от времени прерываемых более короткими последовательностями нулевых бит;

- информация в файле не обязательно непрерывна. Иногда данные внезапно прерываются на границе блока, но продолжаются где-то в другом месте файла;
- некоторые блоки данных в файле встречаются повторно.

Листинг 1.22. Подраздел CodeView версии NB10

```
typedef struct _CV_NB10            // ссылка на данные PDB
      {
      CV_HEADER Header;
      DWORD          dSignature;   // секунд, прошедших
                                   // с 01-01-1970
      DWORD          dAge;         // 1++
      BYTE       abPdbName [];     // завершается нулем
      }
      CV_NB10, *PCV_NB10, **PPCV_NB10;

#define CV_NB10_ sizeof (CV_NB10)
```

Через некоторое время я наконец понял, что это типичные характеристики составного (compound) файла. Составной файл — это небольшая файловая система, упакованная в одном файле. Слова «файловая система» легко объясняют некоторые замеченные ранее факты:

- файловая система делит диск на *сектора* (sectors) фиксированного размера и объединяет сектора в *файлы* (files) переменного размера. Содержащие файл сектора могут быть расположены в любом месте диска и не обязаны быть непрерывными — соответствие секторов файлу определено в каталоге файлов (file directory);
- составной файл делит необработанный (raw) файл на диске на *страницы* (pages) фиксированной длины и группирует эти страницы в *потоки* (streams) переменной длины. Представляющие файл страницы могут быть расположены в любом месте необработанного файла и не обязаны быть непрерывными — соответствие страниц потоку определено в *каталоге потоков* (stream directory).

Понятно, что почти все утверждения касательно файловых систем могут быть применены и к составным файлам простой заменой «сектора» на «страницу» и «файла» на «поток». Модель файловой системы объясняет, почему файл PDB организован в блоках фиксированной длины. Она также объясняет, почему блоки не должны всегда быть непрерывны. Как насчет страниц со скоплениями единичных бит? Такая схема данных очень часто встречается в файловых системах. Для того чтобы отслеживать занятые и свободные сектора на диске, многие файловые системы поддерживают битовый массив использования секторов, каждый бит которого соответствует сектору (или кластеру секторов). Установленный бит соответствует свободному сектору. Когда файловая система выделяет под файл дисковое пространство, она ищет свободные сектора, просматривая биты использования. Когда сектор выделяется под файл, его бит использования сбрасывается в ноль. Та же схема работает и для страниц и потоков составного файла. Длительные последовательности равных единице бит представляют неиспользуемые страницы, а биты, равные нулю, соответствуют существующим потокам.

Осталось объяснить еще один факт — повторение в файле PDB блоков данных. То же происходит и с секторами на диске. Когда в файловой системе файл переписывается несколько раз, для каждой операции записи могут использоваться разные сектора. Таким образом, может получиться так, что на диске будут расположены считающиеся свободными сектора, в которых содержатся старые копии информации из файла. Для файловой системы это не проблема, поскольку если сектор помечен как свободный в битовом массиве использования, то не имеет значения, какие данные он содержит. Как только сектор будет выделен для другого файла, эти данные в любом случае будут перезаписаны. Если снова применить модель файловой системы к составным файлам, это будет означать, что одинаковые страницы остались от предыдущих версий потока, который затем был перезаписан в другие страницы составного файла. Эти данные можно без какого-либо ущерба не принимать во внимание, нам нужно учитывать только те страницы, на которые ссылается каталог потоков. Остальные страницы можно рассматривать как мусор.

Зная основной принцип организации файлов PDB, мы можем перейти к более интересной задаче — изучению их основных строительных блоков. В листинге 1.23 показана схема заголовка PDB. Структура PDB_HEADER начинается с длинной сигнатуры, определяющей версию PDB в виде текстовой строки. Текст завершается символом конца файла EOF (end of file, код ASCII 0x1A) и сопровождается загадочным числом 0x0000474A, или «JG\0\0», если его интерпретировать как строку. Возможно, это инициалы проектировщика формата PDB. Благодаря встроенному символу EOF файлы PDB обладают одним удобным свойством: несведущий пользователь может выполнить в окне консоли команду наподобие type ntoskrnl.pdb, и это не будет сопровождаться выводом мусора на экран. Отобразится лишь сообщение Microsoft C/C++ program database 2.00\r\n. Все файлы идентификаторов Windows 2000 поставляются в формате PDB 2.00. Предположительно, существует также и формат PDB 1.00, но он, по-видимому, организован совсем по-другому.

Листинг 1.23. Заголовок файла PDB

```
#define PDB_SIGNATURE_200 \
          "Microsoft C/C++ program database 2.00\r\n\x1AJG\0"

#define PDB_SIGNATURE_TEXT 40

//  ------------------------------------------------------

typedef struct _PDB_SIGNATURE
      {
      // PDB_SIGNATURE_nnn
      BYTE abSignature [PDB_SIGNATURE_TEXT+4];
      }
      PDB_SIGNATURE, *PPDB_SIGNATURE, **PPPDB_SIGNATURE;

#define PDB_SIGNATURE_ sizeof (PDB_SIGNATURE)

//  ------------------------------------------------------

#define PDB_STREAM_FREE -1
```

```
//  ------------------------------------------------------

typedef struct _PDB_STREAM
      {
      DWORD dStreamSize;    // в байтах. -1 = свободный поток
      PWORD pwStreamPages; // массив номеров страниц
      }
      PDB_STREAM, *PPDB_STREAM, **PPPDB_STREAM;

#define PDB_STREAM_ sizeof (PDB_STREAM)

//  ------------------------------------------------------

#define PDB_PAGE_SIZE_1K       0x0400    // байт на страницу
#define PDB_PAGE_SIZE_2K       0x0800
#define PDB_PAGE_SIZE_4K       0x1000

#define PDB_PAGE_SHIFT_1K 10             // log2 (PDB_PAGE_SIZE_*)
#define PDB_PAGE_SHIFT_2K 11
#define PDB_PAGE_SHIFT_4K 12

#define PDB_PAGE_COUNT_1K 0xFFFF         // номер страницы <
                                         // PDB_PAGE_COUNT_*
#define PDB_PAGE_COUNT_2K 0xFFFF
#define PDB_PAGE_COUNT_4K 0x7FFF

//  ------------------------------------------------------

typedef struct _PDB_HEADER
      {
      PDB_SIGNATURE  Signature;          // PDB_SIGNATURE_200
      DWORD              dPageSize;      // 0x0400, 0x0800, 0x1000
      WORD           wStartPage;         // 0x0009, 0x0005, 0x0002
      WORD           wFilePages;         // размер файла /
                                         // dPageSize
      PDB_STREAM     RootStream;         // каталог потоков
      WORD           awRootPages [];     // страницы, содержащие
                                         // PDB_ROOT
      }
      PDB_HEADER, *PPDB_HEADER, **PPPDB_HEADER;

#define PDB_HEADER_ sizeof (PDB_HEADER)
```

По смещению 0x2C следом за сигнатурой расположен член dPageSize типа DWORD, определяющий размер страницы составного файла в байтах. Допустимы значения 0x400 (1 Кбайт), 0x800 (2 Кбайт) и 0x100 (4 Кбайт). Член wFilePages отражает общее количество страниц, используемых образом файла PDB. При умножении этого значения на размер страницы всегда будет получаться точный размер файла в байтах. wStartPage — это номер страницы (нумерация с нуля), указывающий на первую страницу данных. Смещение в байтах этой страницы можно вычислить, умножив номер страницы на ее размер. Стандартные значения — 9 для страниц размером 1 Кбайт (смещение в байтах 0x2400), 5 для страниц размером 2 Кбайт (смещение в байтах 0x2800) или 2 для страниц в 4 Кбайт (смещение в байтах 0x2000). Страницы между PDB_HEADER и первой страницей данных зарезервиро-

ваны для битового массива использования составного файла, который всегда начинается с начала второй страницы. Это значит, что файл PDB выделяет 0x2000 байт с 0x10000 бит использования в случае, если размер страницы равен 1 или 2 Кбайт, и 0x1000 байт с 0x8000 бит использования, если размер страницы равен 4 Кбайт. Отсюда следует, что файл PDB может содержать масимально 64 Мбайт данных в режиме страниц в 1 Кбайт и 128 Мбайт данных в режиме страниц объемом 2 или 4 Кбайт.

Завершающие структуру PDB_HEADER члены RootStream и awRootPages[] описывают расположение каталога потоков в файле PDB. Как уже замечалось, файл PDB по смыслу представляет собой набор потоков переменной длины, содержащих действительные данные. Расположение потоков управляется в одном каталоге потоков. Может показаться странным, но сам каталог потоков также хранится в потоке, который я назвал «корневым потоком» (root stream). Корневой поток с каталогом потоков может находиться в любом месте файла PDB. Его расположение и размер содержатся в членах RootStream и awRootPages[] структуры PDB_HEADER. Член dStreamSize подструктуры PDB_STREAM определяет количество страниц, занимаемых каталогом потоков, а элементы массива awRootPages[] указывают на страницы с данными.

Давайте рассмотрим простой пример. Структура PDB_HEADER файла ntoskrnl.pdb показана в примере 1.9, нужные нам значения выделены полужирным шрифтом. Видно, что файл PDB использует страницы размером 0x400 байт и содержит 0x02D1 страницу, что дает размер файла в 0xB4400 (738 304 в десятичной системе). Проверка командой dir показывает, что это значение верно. Размер корневого потока равен 0x5B0 байт. С учетом того что размер страницы равен 0x400 байт, это означает, что массив awRootPages[] состоит из двух элементов, расположенных по файловым смещениям 0x3C и 0x3E. В этих элементах содержатся номера страниц, которые нужно умножить на размер страницы, чтобы получить соответствующие смещения в байтах. В нашем случае получаются значения 0xB2000 и 0xB2800.

Основной результат этих вычислений состоит в том, что каталог потоков файла PDB модуля ntoskrnl.exe содержится в двух страницах файла, находящихся соответственно с адреса 0xB2000 по 0xB23FF и с 0xB2800 по 0xB29AF. Частично эти диапазоны показаны в примере 1.10.

Пример 1.9. Пример заголовка PDB

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 01234567B9ABCDEF
---------|-------------------------:-------------------------|-----------------
00000000 | 4D 69 63 72-6F 73 6F 66 : 74 20 43 2F-43 2B 2B 20 | Microsoft C/C++.
00000010 | 70 72 6F 67-72 61 6D 20 : 64 61 74 61-62 61 73 65 | program database
00000020 | 20 32 2E 30-30 0D 0A 1A : 4A 47 00 00-00 04 00 00 |  2.00...JG......
00000030 | 09 00 D1 02-B0 05 00 00 : 5C 00 78 00-C8 02 CA 02 | ..C.°...\.x.И.К.
```

Пример 1.10. Пример каталога потоков PDB (частично)

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 01234567B9ABCDEF
---------|-------------------------:-------------------------|-----------------
000B2000 | 0B 00 00 00-B0 05 00 00 : 98 22 28 00-3A 00 00 00 | ....°...?"(.:...
000B2010 | 88 57 26 00-3B 00 00 00 : 78 57 26 00-A9 02 04 00 | ?W&.B...xW&.©...
000B2020 | F8 BA E9 00-00 00 00 00 : 68 57 26 00-04 40 00 00 | шей.....hW&..@..
000B2030 | C8 29 28 00-B4 9E 01 00 : 08 90 ED 00-3C DF 04 00 | И)(.'?...•н.<Я...
000B2040 | 08 BD E9 00-12 00 C9 02 : C7 02 13 00-C6 02 C6 01 | .Sй...Й.З...Ж.Ж.
000B2050 | C7 01 C8 01-C9 01 CA 01 : CB 01 CC 01-CD 01 CE 01 | З.И.Й.К.Л.М.Н.О.
...
```

```
000B23A0 | BD 00 BE 00-BF 00 C0 00 : C1 00 C2 00-C3 00 C4 00 | S.s.ї.А.Б.В.Г.Д.
000B23B0 | C5 00 C6 00-C7 00 C8 00 : C9 00 CA 00-CB 00 CC 00 | Е.Ж.З.И.Й.К.Л.М.
000B23C0 | CD 00 CE 00-CF 00 D0 00 : D1 00 D2 00-D3 00 D4 00 | Н.О.П.Р.С.Т.У.Ф.
000B23D0 | D5 00 D6 00-D7 00 D8 00 : D9 00 DA 00-DB 00 DC 00 | Х.Ц.Ч.Ш.Щ.Ъ.Ы.Ь.
000B23E0 | DD 00 DE 00-DF 00 E0 00 : E1 00 E2 00-E3 00 E4 00 | _.Ю.Я.а.б.в.г.д.
000B23F0 | E5 00 E6 00-E7 00 EB 00 : E9 00 EA 00-EB 00 EC 00 | е.ж.з.и.й.к.л.м.
---------|-----------------------:-----------------------|----------------
000B2800 | ED 00 EE 00-EF 00 F0 00 : F1 00 F2 00-F3 00 F4 00 | н.о.п.р.с.т.у.ф.
000B2810 | F5 00 F6 00-F7 00 F8 00 : F9 00 FA 00-FB 00 FC 00 | х.ц.ч.ш.щ.ъ.ы.ь.
000B2820 | FD 00 FE 00-FF 00 00 00 : 01 01 02 01-03 01 04 01 | э ю я...........
000B2830 | 05 01 06 01-07 01 08 01 : 09 01 0A 01-0B 01 0C 01 | ................
000B2840 | 0D 01 0E 01-0F 01 10 01 : 11 01 12 01-13 01 14 01 | ................
000B2850 | 15 01 16 01-17 01 18 01 : 19 01 1A 01-1B 01 1C 01 | ................
...
000B2950 | 95 01 96 01-97 00 98 01 : 99 01 9A 01-9B 01 9C 01 | ?.?.?.?.?.?.?.?.
000B2960 | 9D 01 9E 01-9F 01 A0 01 : A1 01 A2 01-A3 01 A4 01 | •.?.?. .Ў.ў.J.¤.
000B2970 | A5 01 A6 01-A7 01 A8 01 : A9 01 AA 01-AB 01 AC 01 | Ґ.¦.§.Ё.©.Є.«.¬.
000B2980 | AD 01 AE 01-AF 01 B0 01 : B1 01 B2 01-B3 01 B4 01 | .®.Ї.°.±.І.і.ґ.
000B2990 | B5 01 B6 01-B7 01 B8 01 : B9 01 BA 01-BB 01 BC 01 | µ.¶.·.ё.№.є.».j.
000B29A0 | BD 01 BE 01-BF 01 C0 01 : C1 01 C2 01-C3 01 C4 01 | S.s.ї.А.Б.В.Г.Д.
```

Каталог потоков состоит из двух разделов: части заголовка в виде структуры PDB_ROOT, как определено в листинге 1.24, и части данных, состоящей из массива 16-битных номеров страниц. Член wCount раздела PDB_ROOT определяет количество потоков, хранимых в составном файле PDB. Массив aStreams[] содержит по элементу PDB_STREAM (листинг 1.23) для каждого потока, а номера страниц расположены сразу же после последнего элемента aStreams[]. В примере 1.10 количество потоков равно восьми, как показывает выделенное значение по смещению 0xB2000. Восемь последующих структур PDB_STREAM определяют потоки размеров 0x5B0, 0x3A, 0x38, 0x402A9, 0x0, 0x4004, 0x19EB4 и 0x4DF3C. Эти значения также выделены в примере 1.10. Если выразить размер потоков в страницах объемом 1 Кбайт, получатся значения 0x2, 0x1, 0x1, 0x101, 0x0, 0x11, 0x68 и 0x138, что дает в общей сложности 0x2B6 страниц, используемых потоками. Первое выделенное значение после массива PDB_STREAM — это первый элемент списка номеров страниц. Считая по два байта на каждый номер страницы и принимая во внимание тот факт, что каталог страниц прерывается одной страницей, принадлежащей кому-то еще, следующее смещение после номеров страниц должно быть 0xB2044 + 0x400 + (0x2B6 * 2) = 0xB29B0, что в точности соответствует рисунку.

Листинг 1.24. Каталог потоков PDB

```
#define  PDB_STREAM_DIRECTORY   0
#define  PDB_STREAM_PDB         1
#define  PDB_STREAM_PUBSYM      7

//  ------------------------------------------------------

typedef struct _PDB_ROOT
      {
      WORD          wCount;         // < PDB_STREAM_MAX
      WORD          wReserved;      // 0
      PDB_STREAM    aStreams [];    // поток #0 зарезервирован
                                    // под таблицу потоков
      }
      PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;

#define PDB_ROOT_ sizeof (PDB_ROOT)
```

Поиск блока номера страницы, сопоставленного заданному потоку, осуществляется довольно замысловатым способом, поскольку каталог страниц не предоставляет никаких данных для поиска, кроме размера потока. Если вас интересует поток 3, нужно вычислить число страниц, занимаемых потоками 1 и 2, чтобы получить требуемый начальный индекс в массиве номеров страниц. После определения списка номеров страниц потока чтение его данных не составляет труда. Надо пройти по списку и умножить каждый номер страницы на размер страницы, что даст смещение от начала файла, и прочитать страницы по полученным смещениям, пока не будет достигнут конец потока. На первый взгляд, разбор файла PDB представлялся достаточно трудным. Однако оказалось, что на самом деле это совсем несложно, вероятно, значительно проще, чем разбор файла .dbg. Составной характер формата PDB и его ясная схема произвольного доступа к страницам потока сводит задачу чтения потока к простому соединению страниц фиксированного размера. Меня лично этот элегантный механизм доступа к данным приводит в восхищение.

Даже еще большие преимущества формата PDB проявляются при обновлении файла PDB. Для вставки данных в файл с последовательной структурой, как правило, требуется сдвигать большие части содержимого. Структура произвольного доступа файлов PDB, позаимствованная у файловых систем, дает возможность добавлять и удалять данные с минимумом усилий, точно так же, как при изменении файлов на носителе с файловой системой. Необходимо только перестраивать каталог потоков, если поток при увеличении или уменьшении размера переходит границу страницы. Это важное свойство облегчает добавочное (incremental) обновление файлов PDB. В статье из Knowledge Base «INFO: PDB и DBG Files — What Are They Are and How They Work» (файлы PDB и DBG — описание и принцип работы) Microsoft утверждает следующее:

«Расширение .PDB означает 'program database' (база данных программы). Это новый формат для хранения отладочной информации, представленный в Visual C++ версии 1.0. В будущем файлы .PDB будут также хранить другую информацию о состоянии проекта. Одним из наиболее существенных факторов изменения формата была потребность в добавочной компоновке (incremental linking) отладочных версий программ, эта возможность впервые появилась в Visual C++ версии 2.0».

После того как мы разобрались с внутренним форматом файлов PDB, нужно понять, как распознавать содержимое потоков. После изучения различных файлов PDB я пришел к выводу, что цели потока с каждым конкретном номером определены заранее. Первый поток, по-видимому, всегда содержит каталог потоков, а второй — информацию для проверки соответствия файла PDB файлу .dbg. Например, второй поток содержит члены `dSignature` и `dAge`, значения которых должны совпадать с значениями соответствующих членов раздела `NB10` CodeView, как показано в листинге 1.22. Восьмой поток представляет наибольший интерес в рамках материала этой главы, поскольку в нем как раз содержится интересующая нас информация об идентификаторах CodeView. Значение остальных потоков пока является для меня загадкой и открывает еще одно направление будущих исследований.

Я не хочу приводить здесь пример программы чтения формата PDB, поскольку это вышло бы за рамки рассматриваемого в главе материала. Но я настоятельно

рекомендую ознакомиться с исходными файлами w2k_img.c и w2k_img.h на компакт-диске с примерами. Ищите в этих файлах функции с именами imgPdb*() и элементы данных с именами PDB_*. Между прочим, на компакт-диске находится полностью готовая программа чтения файлов PDB и приведены все ее исходные коды. Вам эта программа уже знакома — это утилита w2k_dump.exe, при помощи которой были созданы приведенные выше шестнадцатеричные снимки. У этой простой утилиты консольного режима есть параметр командной строки +p, который дает возможность разбора потока PDB. Если заданный файл не является корректным файлом PDB, программа переходит в обычный режим последовательного шестнадцатеричного снимка. Файлы проекта Visual C/C++ программы w2k_dump.exe находятся на компакт-диске в каталоге \src\w2k_dump.

Идентификаторы PDB

После такого длительного и, надеюсь, интересного обзора формата PDB, давайте вернемся к нашей исходной цели: получение символьной информации CodeView. К счастью, это задача похожа на задачу последовательного перебора открытых идентификаторов в подразделе NB09 CodeView. Определив месторасположение содержащего идентификаторы потока, мы снова видим последовательность похожих на OMF записей переменной длины. К сожалению, форматы записей NB09 и NB10 несколько отличаются, однако отклонения не такие уж существенные. В листинге 1.25 показана схема структуры PDB_PUBSYM. По сравнению с соответствующей структурой CV_PUBSYM формата NB09 (листинг 1.21) наиболее значительны следующие отличия: члены dOffset и wSegment переместились несколько ближе к концу, а значение тега идентификаторов PDB теперь равно 0x1009, а не 0x203.

Листинг 1.25. Структура PDB_PUBSYM

```
#define PDB_PUB32 0x1009

//  ----------------------------------------------------

typedef struct _PDB_PUBSYM
      {
      OMF_HEADER     Header;
      DWORD          dReserved;
      DWORD          dOffset;
      WORD           wSegment;      // 1-индекс раздела,
                                    // начинающийся с 1
      OMF_NAME       Name;          // дополнено нулями до
                                    // следующего DWORD
      }
      PDB_PUBSYM,  *PPDB_PUBSYM,  **PPPDB_PUBSYM;

#define PDB_PUBSYM_ sizeof (PDB_PUBSYM)

#define PDB_PUBSYM_SIZE(_p) \
      ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define PDB_PUBSYM_NEXT(_p) \
      ((PPDB_PUBSYM) ((PBYTE) (_p) + PDB_PUBSYM_SIZE (_p)))
```

При помощи объединения IMG_PUBSYM удобно работать с записями идентификаторов независимо от их типа. Объединение можно трактовать тремя способами:

1. OMF_HEADER: так следует рассматривать объединение, пока не станет известен тип идентификатора. Информация заголовка дает возможность определить тип или перейти к следующей записи.
2. CV_PUBSYM: такая интерпретация корректна, только если член wRecordType структуры-заголовка OMF_HEADER установлен в значение CV_PUB32 (0x203).
3. PDB_PUBSYM: эта интерпретация корректна, только если член wRecordType структуры-заголовка OMF_HEADER установлен в значение PDB_PUB32 (0x1009).

В конце листинга 1.26 определены макросы IMG_PUBSYM_SIZE() и IMG_PUBSYM_NEXT(), позволяющие независимо от типа определить соответственно размер текущей записи и адрес следующей за ней.

Листинг 1.26. Объединение IMG_PUBSYM

```
typedef union _IMG_PUBSYM
      {
      OMF_HEADER Header;            // CV_PUB32 or PDB_PUB32
      CV_PUBSYM  CvPubSym;
      PDB_PUBSYM PdbPubSym;
      }
      IMG_PUBSYM, *PIMG_PUBSYM, **PPIMG_PUBSYM;

#define IMG_PUBSYM_ sizeof (IMG_PUBSYM)

#define IMG_PUBSYM_SIZE(_p) \
      ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define IMG_PUBSYM_NEXT(_p) \
      ((PIMG_PUBSYM) ((PBYTE) (_p) + IMG_PUBSYM_SIZE (_p)))
```

Вычисление адреса идентификатора

Члены wSegment и dOffset записей идентификаторов CV_PUBSYM и PDB_PUBSYM вместе с массивом IMAGE_SECTION_HEADER в начале файла .dbg предоставляют необходимую информацию для вычисления адреса идентификатора относительно базового адреса модуля. Если файл .dbg не содержит каких-либо данных OMAP в подразделах IMAGE_DEBUG_TYPE_OMAP_TO_SRC и IMAGE_DEBUG_TYPE_OMAP_FROM_SRC, алгоритм вычисления адреса несложен:

- прочитать значение wSegment записи идентификатора и уменьшить его на единицу;
- найти по полученному индексу заголовок IMAGE_SECTION_HEADER раздела, в котором расположен идентификатор;
- получить значение VirtualAddress этого заголовка IMAGE_SECTION_HEADER;
- добавить к записи идентификатора значение dOffset.

В том случае, если известен адрес загрузки модуля, для получения абсолютного линейного адреса идентификатора достаточно просто прибавить вычисленный

относительный адрес к базовому. Член ImageBase заголовка IMAGE_SEPARATE_DEBUG_HEADER в самом начале файла .dbg задает предпочтительный адрес загрузки модуля. К сожалению, этот адрес не сильно нам поможет, потому что многие модули ядра в действительности загружаются по совершенно другим адресам. Например, в файле ntoskrnl.dbg содержится предпочтительный адрес загрузки 0x00400000, что, разумеется, неверно, поскольку этот адрес выходит далеко за диапазон адресов памяти ядра. Поэтому библиотека w2k_img.dll предоставляет функцию API imgModuleBase(), которая пытается определить нахождение модуля ядра в памяти. Эта функция использует недокументированную функцию NtQuerySystemInformation(), экспортируемую модулем ntdll.dll, чтобы получить список найденных в памяти модулей. Однако эта функция работает для Windows 2000/NT. Для совместимости с Windows 9x функция imgModuleBase() загружает ntdll.dll динамически, поэтому в этих операционных системах w2k_img.dll не будет выдавать ошибку динамической компоновки сразу при загрузке, а функция всегда будет возвращать NULL-указатель. То же значение вы получите и в Windows 2000, и в Windows NT 4.0, если указанный модуль отсутствует в памяти.

Преобразование адресов OMAP

В нескольких файлах идентификаторов Windows 2000 содержатся подразделы OMAP, которым соответствуют элементы IMAGE_DEBUG_DIRECTORY с типами идентификаторов IMAGE_DEBUG_TYPE_OMAP_TO_SRC и IMAGE_DEBUG_TYPE_OMAP_FROM_SRC. Данные OMAP — еще одна недокументированная особенность средств разработки, поэтому об их назначении пока существуют только предположения. Данные OMAP внутри файла .dbg состоят из двух массивов структур OMAP_TO_SRC и OMAP_FROM_SRC (см. листинг 1.27), эта информация используется для вычисления адресов идентификаторов по значениям смещений, хранимых в записях CV_PUBSYM или PDB_PUBSYM.

В одной из своих превосходных статей в *MSJ* из серии «Under the Hood», посвященных отладочной информации, Матт Питрек пишет об OMAP следующее:

«OMAP, еще один сравнительно новый вид отладочной информации, не документирован, за исключением немногочисленных туманных упоминаний в файле WINNT.H и справочной документации Win32 SDK. Вероятно, во время внутренней процедуры сборки Microsoft небольшие фрагменты кода из исполняемых файлов EXE и библиотек DLL перемещаются в другое место, чтобы наиболее часто используемый код был расположен в начале раздела кода. Предположительно, это сохраняет минимальный размер рабочего множества процесса. При перемещении блоков кода соответствующая отладочная информация не обновляется, а создается информация OMAP. Она позволяет преобразовывать исходный адрес из таблицы идентификаторов в измененный адрес, по которому переменная или строка кода в действительности находится в памяти, а также осуществлять обратное преобразование».

Листинг 1.27. Записи таблицы OMAP_TO_SRC и OMAP_FROM_SRC

```
typedef struct _OMAP_TO_SRC
      {
      DWORD dTarget;
      DWORD dSource;
      }
      OMAP_TO_SRC, *POMAP_TO_SRC, **PPOMAP_TO_SRC;

#define OMAP_TO_SRC_ sizeof (OMAP_TO_SRC)

//  ------------------------------------------------------

typedef struct _OMAP_FROM_SRC
      {
      DWORD dSource;
      DWORD dTarget;
      }
      OMAP_FROM_SRC, *POMAP_FROM_SRC, **PPOMAP_FROM_SRC;

#define OMAP_FROM_SRC_ sizeof (OMAP_FROM_SRC)
```

Более чем через два года ведущий раздела в журнале *MSJ* Джон Роббинс (John Robbins) уточняет это предположение в статье «Bugslayer» в номере за октябрь 1997 г.

«Представляет интерес недокументированная информация OMAP, поскольку, по-видимому, она имеет некоторое отношение к перемещению блоков кода. (Мой коллега по MSJ *Матт Питрек кратко обсуждал эту тему в своем разделе «Under the Hood» за май 1997 г.) Я предполагаю, что Microsoft использует какой-то инструмент, пакующий двоичный файл так, что наиболее часто используемый код перемещается в начало, а остальной код — в конец, что существенно уменьшает рабочее множество. Следовательно, это переупорядочивание двоичного файла ускоряет работу программы, поскольку уменьшает количество загружаемых страниц виртуальной памяти с ее кодом».*

Хотя аргумент об уменьшении рабочего множества выглядит разумным, это расходится с тем фактом, что в модуле ntoskrnl.exe очень часто используются данные OMAP. Как я покажу в главе 4, модуль ntoskrnl.exe целиком отображается в одну страницу объемом 4 Мбайт, которая постоянно находится в памяти, и поэтому деление кода на более часто и менее часто используемые фрагменты не даст никаких преимуществ за счет уменьшения подкачки страниц. Я считаю, что это деление производится, чтобы способствовать предвыборке команд (instruction prefetch) процессора. Исследование таблиц OMAP показывает, что содержащиеся там адреса, как правило, указывают на начало функции, на инструкцию, следующую сразу после выполнения инструкции jump или call, или на неиспользуемый заполняющий код. Из этого можно сделать вывод, что данные OMAP используются для переключения ветвей инструкций if/else. Несомненно, разработчики ядра Windows 2000 из Microsoft могут каким-то образом сообщить компилятору, какая из ветвей, if или else, выполняется чаще, поэтому менее часто выполняемый участок кода можно переместить в другое место. Как правило, компилятор стремится поместить код функции в единый монолитный блок и не разделяет ветви if/else.

Легко заметить, однако, что в модулях ядра Windows 2000 большие функции с многочисленными операторами if/else сильно фрагментированы. Тот факт, что неделимые участки кода OMAP соответствуют ветвям условного оператора, навел меня на мысль, что данные OMAP как-то связаны с предугадыванием ветвления. Если менее часто выполняемые ветви отделены от более часто выполняемых, процессор может осуществлять предвыборку команд более эффективно.

Таблица OMAP_TO_SRC преобразовывает действительное смещение инструкции в исходное смещение, например действительное смещение функции API ExInterlockedAddLargeInteger() относительно базового адреса модуля ntoskrnl.exe равно 0x0000231E. Для проверки этого утверждения введите команду u ExInterlockedAddLargeInteger в командной строке Kernel Debugger — он выведет несколько дизассемблированных строк, начиная с линейного адреса 0x8040231E. Если вычесть из этого значения адрес загрузки ntoskrnl.exe 0x80400000, то, как и ожидалось, мы получим значение 0x0000231E. Если просмотреть таблицу OMAP_TO_SRC внутри ntoskrnl.dbg, мы найдем запись, у которой член dTarget равен этому смещению, а соответствующее смещение dSource равно 0x0005E7E4. Функция ExInterlockedAddLargeInteger() расположена в разделе .text, а смещение этого раздела относительно базового адреса образа равно 0x000004C0 в соответствии с его заголовком IMAGE_SECTION_HEADER. Если вычесть смещение раздела из исходного смещения, получится необработанное смещение идентификатора 0x0005E324, и именно это значение принимает член dOffset записи PDB_PUBSYM, определяющий идентификатор ExInterlockedAddLargeInteger. Выглядит не очень сложным, не правда ли? Но на самом деле все не так просто.

Записи OMAP_TO_SRC всегда отсортированы в восходящем порядке по отношению к адресу назначения. Это неплохая идея, поскольку это дает возможность ускорить поиск адресов, применив алгоритм двоичного поиска. Таблица OMAP_FROM_SRC по существу является точной копией таблицы OMAP_TO_SRC, только в ней переставлены местами исходные адреса и адреса назначения и записи отсортированы уже по исходному адресу. Поддержка двух таблиц позволяет легко преобразовывать адреса в обоих направлениях.

Данные OMAP обладают одной особенностью, озадачившей меня на несколько дней: при преобразовании исходных адресов в адреса назначения через таблицу OMAP_FROM_SRC нельзя непосредственно использовать значения VirtualAddress, хранимые в массиве IMAGE_SECTION_HEADER файла .dbg. Для всех разделов PE, кроме первого, будут получаться слишком большие адреса назначения. Причина в том, что значения VirtualAddress корректны только с точки зрения адреса назначения. На стороне исходных адресов применяются различные адреса разделов. Теперь основная задача — найти исходные адреса разделов PE. После неоднократного, но безуспешного просмотра файлов .dbg и .pdb в поисках каких-либо таблиц, которые могли бы осуществлять преобразование, я наконец нашел способ, дающий прекрасные результаты, хотя я не до конца уверен в его правильности. Чтобы определить исходный адрес раздела, я просто перенумеровал все записи OMAP_TO_SRC, принадлежащие к этому разделу, и нашел минимальный исходный адрес среди адресов этих записей. Эта процедура основана на том предположении, что OMAP

является просто перестановкой фрагментов кода, поэтому фрагмент с минимальным исходным адресом — это именно тот фрагмент, который был помещен в самое начало раздела. Я применял эту технику к многочисленным файлам идентификаторов Windows 2000, и до сих пор она меня не подводила.

Если вам захотелось реализовать синтаксический анализатор файлов идентификаторов, основываясь на вышеизложенной информации, — так и сделайте! Можно также использовать без изменений библиотеку w2k_img.dll с компакт-диска с примерами или воспользоваться ее исходным кодом. Эта DLL содержит все необходимое для разбора файлов .dbg и .pdb, а также многое другое. Наиболее действенный экспортируемый ею набор функций API — это группа из трех функций imgTable*() (табл. 1.5), прототипы которых показаны в листинге 1.28. Они предназначены в помощь разработчикам отладчиков и дизассемблеров. При помощи функции API imgTableLookup() приложение может отображать идентификаторы вместо «сырых» адресов, а на основе функции imgTableResolve() можно реализовать поиск идентификаторов. Обе функции тщательно оптимизированы по скорости, что очень важно для приложений, просматривающих символьную информацию большого объема. Представленная ниже программа просмотра (браузер) идентификаторов основана на библиотеке w2k_img.dll. Эта программа-пример способна записывать в файл снимок памяти отсортированного списка всех идентификаторов ntoskrnl.exe вместе с большим количеством дополнительной информации менее чем за 2 секунды.

Таблица 1.5. Функции управления таблицей идентификаторов, экспортируемые модулем w2k_img.dll

Имя	Описание
imgTableLoad()	Создает таблицу идентификаторов TABLE на основе файла .dbg или .pdb
imgTableLookup()	Ищет запись IMG_ENTRY в таблице идентификаторов с совпадающим адресом идентификатора
imgTableResolve()	Ищет запись IMG_ENTRY в таблице идентификаторов с совпадающим именем идентификатора

Листинг 1.28. Прототипы функции управления таблицей идентификаторов

```
PIMG_TABLE WINAPI imgTableLoad (PBYTE  ptPath,
                                PVOID  pBase);

PIMG_ENTRY WINAPI imgTableLookup (PIMG_TABLE  pit,
                                  PVDID       pAddress,
                                  PDWORD      pdOffset);

PIMG_ENTRY WINAPI imgTableResolve (PIMG_TABLE  pit,
                                   PBYTE       pbSymbol);
```

В листинге 1.29 приведены структуры, с которыми работают функции imgTable*(). Очевидно, они не похожи на обсуждаемые выше структуры CodeView и PDB. На самом деле функции управления таблицей идентификаторов из библиотеки w2k_img.dll полностью переупорядочивают информацию из файлов идентификаторов для упрощения и ускорения ее обработки. Фундаментальной структурой является IMG_TABLE, содержащая всю символьную информацию. Она образована

из заголовка фиксированного размера, массива структур IMG_ENTRY и их массивов IMG_INDEX. Поскольку размер массивов меняется в зависимости от количества идентификаторов, IMG_TABLE также содержит три указателя на базовые адреса IMG_INDEX. Как свидетельствуют комментарии в листинге 1.29, индексы отсортированы по адресу, по имени с учетом регистра и по имени без учета регистра. Такая организация индексов удобна не только для приложений, выводящих списки идентификаторов, но также и для функций imgTableLookup() и imgTableResolve(), поскольку на основе этих индексов функции могут осуществлять двоичный поиск адресов и имен.

Особенно удобно, что структура IMG_ENTRY содержит информацию о соглашении о вызовах идентификатора, которая извлекается непосредственно из декорированных идентификаторов на основе правил из табл. 1.4. С этой нетривиальной задачей справляется функция imgSymbolUndecorate(), показанная в листинге 1.30. Сначала она пытается определить один из общих префиксов, перечисленных в массиве apbPrefixes[]. На следующем шаге ищется содержащая размер стека завершающая часть идентификатора, состоящая из символа @ и десятичного числа. Соглашение о вызовах определяется путем проверки определенных сочетаний префикс-завершение. w2k_img.dll очень надежно осуществляет обратное преобразование декорированных идентификаторов. Она правильно обрабатывает все шлюзы импорта __fastcall, на которых спотыкается imagehlp.dll. Функция imgSymbolUndecorate(), тем не менее, не пытается преобразовывать декорированные идентификаторы C++ и PCH. Может быть, я добавлю эту возможность в будущую версию w2k_img.dll.

Листинг 1.29. Структуры для управления таблицей идентификаторов

```
#define  IMG_CONVENTION_UNDEFINED    0
#define  IMG_CONVENTION_STDCALL      1
#define  IMG_CONVENTION_CDECL        2
#define  IMG_CONVENTION_FASTCALL     3

//  ------------------------------------------------------

typedef struct _IMG_ENTRY
      {
      DWORD dSection;               // номер раздела
                                    // (начиная с 1)
      PVOID pAddress;               // адрес идентификатора
      DWORD dConvention;            // соглашение о вызовах
                                    // IMG_CONVENTION_*
      DWORD dStack;                 // количество байт
                                    // стека аргументов
      BOOL  fExported;              // TRUE, если экспортируемый
                                    // идентификатор
      BOOL  fSpecial;               // TRUE, если специальный
                                    // идентификатор
      BYTE  abSection [IMAGE_SIZEOF_SHORT_NAME+4]; // имя раздела
      BYTE  abSymbol [256];         // недекорированное имя
                                    // идентификатора
      BYTE  abDecorated [256];      // декорированное имя
                                    // идентификатора
```

продолжение

Листинг 1.29 (*продолжение*)

```
    }
    IMG_ENTRY, *PIMG_ENTRY, **PPIMG_ENTRY;

#define IMG_ENTRY_ sizeof (IMG_ENTRY)

//  ------------------------------------------------------

typedef struct _IMG_INDEX
    {
    PIMG_ENTRY apEntries [1];
    }
    IMG_INDEX, *PIMG_INDEX, **PPIMG_INDEX;

#define IMG_INDEX_ sizeof (IMG_INDEX)
#define IMG_INDEX__(_n) ((_n) * IMG_INDEX_)

//  ------------------------------------------------------

typedef struct _IMG_TABLE
    {
    DWORD          dSize;          // размер таблицы в байтах
    DWORD          dSections;      // количество разделов
    DWORD          dSymbols;       // количество
                                   // идентификаторов
    DWORD          dTimeStamp;     // временная метка модуля
                                   // (секунд, прошедших с
                                   //  1.1.1970)
    DWORD          dCheckSum;      // контрольная сумма модуля
    PVOID          pBase;          // базовый адрес модуля
    PIMG_INDEX     piiAddress;     // записи, отсортированные
                                   // по адресу
    PIMG_INDEX     piiName;        // записи, отсортированные
                                   // по имени
    PIMG_INDEX     piiNameIC;      // записи, отсортированные
                                   // по имени (без учета
                                   // регистра)
    BOOL           fUnicode;       // формат символов
    union
        {
        TBYTE atPath [MAX_PATH]; // путь файла .dbg
        BYTE  abPath [MAX_PATH]; // путь файла .dbg (ANSI)
        WORD  awPath [MAX_PATH]; // путь файла .dbg
                                 // (Unicode)
        };
    IMG_ENTRY  aEntries [];        // массив информации об
                                   // идентификаторах
    }
    IMG_TABLE, *PIMG_TABLE, **PPIMG_TABLE;

#define IMG_TABLE_ sizeof (IMG_TABLE)

#define IMG_TABLE__(_n) \
    (IMG_TABLE_ + ((_n) * IMG_ENTRY_) + \
    (3 * IMG_INDEX__ (_n)))
```

Листинг 1.30. Функция API imgSymbolUndecorate()

```
DWORD WINAPI imgSymbolUndecorate (PBYTE   pbSymbol,
                                  PBYTE   pbBuffer,
                                  PDWORD  pdConvention)
{
PBYTE apbPrefixes [] = {"__imp__", "__imp_@",
                                   "__imp_", "_",
                                   "@", "\x7F",
                                   NULL};

BYTE  abBuffer [256] = "";
DWORD i, j, k, l;
DWORD dConvention = IMG_CONVENTION_UNDEFINED;
DWORD dStack      = -1;

if (pbSymbol != NULL)
    {
    // пропустить общие префиксы
    for (i = j = 0; apbPrefixes [i] != NULL; i++)
        {
        for (j = 0; apbPrefixes [i] [j]; j++)
            {
            if (apbPrefixes [i] [j] != pbSymbol [j])
                break;
            }
        if (!apbPrefixes [i] [j]) break;
        j = 0;
        }
    // проверить наличие нескольких '@'
    for (k = j, l = 0; (l < 2) && pbSymbol [k]; k++)
        {
        if (pbSymbol [k] == '@') l++;
        }
    // не преобразовывать в случае нескольких '@'
    // или идентификатора C++
    if ((l == 2) || (pbSymbol [0] == '?'))
        {
        j = 0;              // сохранить префикс
        k = MAXDWORD;       // сохранить длину
        }
    else
        {
        // найти следующий '@'
        for (k = j;
             pbSymbol [k] && (pbSymbol [k] != '@');
             k++);

        // если '@' найден, прочитать количество
        // байт стека аргументов
        if (pbSymbol [k] == '@')
            {
            dStack = 0;

            for (l = k + 1;  (pbSymbol [l] >= '0') &&
                             (pbSymbol [l] <= '9'); l++)
                {
                dStack *= 10;
```

продолжение

Листинг 1.30 (*продолжение*)

```
                    dStack += pbSymbol [l] - '0';
                    }
                // не преобразовывать при нечисловой или
                // пустой завершающей части
                if (pbSymbol [l] || (l == k + 1))
                    {
                    dStack = -1;        // нет информации о
                                        // размере стека

                    j = 0;              // сохранить префикс
                    k = MAXDWORD;       // сохранить длину
                    }
                }
            }
        // в случае односимвольного префикса определить
        // соглашение о вызовах
        if (j == 1)
            {
            switch (pbSymbol [0])
                {
                case '@':
                    {
                    dConvention = IMG_CONVENTION_FASTCALL;
                    break;
                    }
                case '_':
                    {
                    dConvention = (dStack != -1
                                    ? IMG_CONVENTION_STDCALL
                                    : IMG_CONVENTION_CDECL);
                    break;
                    }
                }
            }
        // скопировать отобранную часть имени
        k = min (k - j, sizeof (abBuffer) - 1);
        lstrcpynA (abBuffer, pbSymbol + j, k + 1);
        }
    if (pbBuffer != NULL)
        {
        lstrcpyA (pbBuffer, abBuffer);
        }
    if (pdConvention != NULL) *pdConvention = dConvention;
    return dStack;
    }
```

Заметьте, что функция imgTableResolve() игнорирует все идентификаторы с неопределенным соглашением о вызовах. Это ограничение исключает из рассмотрения все идентификаторы шлюзов импорта, C++ и PCH. К сожалению, при этом также исключаются и «хорошие» идентификаторы, не декорированные по стандартной схеме. Я не думаю, правда, что это большая проблема, поскольку такие идентификаторы используются нечасто.

Основная структура клиентского приложения для библиотеки w2k_img.dll приведена в листинге 1.31. Сначала приложение при помощи функции imgTableLoad()

загружает таблицу идентификаторов из файла .dbg, заданного в аргументе ptPath. Если файл содержит подраздел CodeView NB10, загружается также соответствующий файл PDB. Если возвращенный указатель корректен, записи идентификаторов можно перенумеровать четырьмя способами, которые описаны в комментариях внутри цикла for(). По существу, клиент может использовать исходный порядок идентификаторов, так, как они появляются в файле .dbg или .pdb, или же выбрать один из предопределенных индексов сортировки. Когда обработка идентификатора завершается, приложение должно уничтожить таблицу идентификаторов, вызвав функцию imgMemoryDestroy(). Вот и все! Приложению не нужны подробности о внутренней организации файлов идентификаторов. Вся необходимая ему информация хранится в структурах IMG_TABLE и IMG_ENTRY, значения которых устанавливаются функцией imgTableLoad().

Листинг 1.31. Работа с функциями управления таблицей идентификаторов

```
VOID WINAPI SymbolProcessor (PTBYTE ptPath)
      (
      PIMG_TABLE     pit;
      PIMG_ENTRY     pie;
      PVOID               pBase;
      DWORD               i;


      pBase = imgModuleBase (ptPath);   // получить адрес
                                        // загрузки текущего
                                        // модуля
      if ((pit = imgTableLoad (ptPath, pBase)) != NULL)
           {
           for (i = j = 0; i < pit->dSymbols; i++)
                {
                // Возможность #1: порядок идентификаторов,
                // заданный по умолчанию
                // pie = pit->aEntries + i;

                // Возможность #2: идентификаторы отсортированы
                // по адресу
                // pie = pit->piiAddress->apEntries [i];

                // Возможность #3: идентификаторы отсортированы
                // по имени (с учетом регистра)
                // pie = pit->piiName->apEntries [i];

                // Возможность #4: идентификаторы отсортированы
                // по имени (без учета регистра)
                // pie = pit->piiNameIC->apEntries [i];

                // Теперь получите указатели на IMG_ENTRY
                // следующего идентификатора!
                // Сделайте с ним что-нибудь полезное...

                }
           imgMemoryDestroy (pit);
           }
      return;
      }
```

Типичное клиентское приложение для w2k_img.dll будет представлено в следующем разделе. Отметьте, что я еще вернусь к этой действенной DLL в главе 6, где она послужит весьма необычной цели: с ее помощью мы будем искать адреса внутренних идентификаторов ntoskrnl.exe, которые и недокументированы, и не экспортируются, а работающая с ней вместе DLL воспользуется этой информацией для вызова функции или чтения значения по этому адресу. Этот трюк кажется необычным, но он прекрасно работает и может решить некоторые трудные проблемы программирования и отладки. Будьте в курсе!

Еще один браузер идентификаторов Windows 2000

Приложение-пример, которое продемонстрирует способ работы с w2k_img.dll, является альтернативной версией представленного в предыдущем разделе браузера идентификаторов. Оно называется w2k_sym2.exe, но, несмотря на схожесть имен, это не просто переработка w2k_sym.exe. Приложения-примеры обладают весьма различными возможностями и параметрами командной строки — сравните их справочные экраны, показанные в примерах 1.5 и 1.11. Исходный код программы w2k_sym2.exe находится на прилагающемся к книге компакт-диске в каталоге \src\w2k_sym2.

Пример 1.12 показывает пример выходных данных, полученных при выполнении команды w2k_sym2 +nu beep.sys. Параметр +n задает сортировку по имени без учета регистра, а параметр +u указывает, что в любом случае нужно включать идентификаторы с неизвестным соглашением о вызовах. Идентификаторы со значениями CDECL или STDCALL в столбце ARGUMENTS относятся к адресам функций или глобальных переменных. Оставшиеся строки в примере 1.12 представлют собой наиболее важные шлюзы импорта из модулей ntoskrnl.exe или hal.dll.

Пример 1.11. Справка по командам w2k_sym2.exe

```
// w2k_sym2.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_sym2 { [+-anNiprdxusz] [:<sections>] [/<symbols>] <module> }

      +    enable subsequent options
      -    disable subsequent options
      a    sort by address
      n    sort by name
      N    sort by name (case sensitive)
      i    ignore case in filter strings
      p    force preferred load address
      r    display relative addresses
      d    display decorated symbols
      x    display exported symbols only
      u    include symbols with unknown calling convention
      s    include special symbols
      z    include zero-address symbols

<sections> and <symbols> are filter expressions,
optionally containing the wildcards * and ?.
```

Пример 1.12. Образец выходных данных программы w2k_sym2.exe

```
// w2k_sym2.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Module name:        beep.sys
Time stamp:         Wednesday, 10-20-1999, 22:18:59
Base address:       0xF09CF000
Check sum:          0x0000C54F
Symbol file:        E:\WINNT\Symbols\sys\beep.dbg
Symbol table:       23520 bytes
Symbol filter:      *
Sections:           *

    # INDEX ADDRESS      SECTION       ARGUMENTS     X NAME
-------------------------------------------------------------
    1     0 F09CF70C   2 .rdata                     _allmul
    2     1 F09CF6B2   1 .text          CDECL       _allmul
    3     2 F09CF7B4   3 INIT           CDECL       _IMPORT_DESCRIPTOR_HAL
    4     3 F09CF7A0   3 INIT           CDECL       _IMPORT_DESCRIPTOR_ntoskrnl
    5     4 F09CF7C8   3 INIT           CDECL       _NULL_IMPORT_DESCRIPTOR
    6     5 F09CF34C   1 .text        8 STDCALL     BeepCancel
    7     6 F09CF39E   1 .text        8 STDCALL     BeepCleanup
    8     7 F09CF50E   1 .text        8 STDCALL     BeepClose
    9     8 F09CF456   1 .text        8 STDCALL     BeepDeviceControl
   10     9 F09CF4C0   1 .text        8 STDCALL     BeepOpen
   11    10 F09CF572   1 .text        8 STDCALL     BeepStartIo
   12    11 F09CF660   1 .text       10 STDCALL     BeepTimeOut
   13    12 F09CF67E   1 .text        4 STDCALL     BeepUnload
   14    13 F09CF29A   1 .text        8 STDCALL     DriverEntry
   15    14 F09CF6C0   2 .rdata       4             ExAcquireFastMutex
   16    15 F09CF6C4   2 .rdata       4             ExReleaseFastMutex
   17    16 F09CF6D4   2 .rdata                     HAL_NULL_THUNK_DATA
   18    17 F09CF6D0   2 .rdata       4             HalMakeBeep
   19    18 F09CF724   2 .rdata       4             InterlockedDecrement
   20    19 F09CF6E0   2 .rdata       4             InterlockedExchange
   21    20 F09CF708   2 .rdata       4             InterlockedIncrement
   22    21 F09CF6E8   2 .rdata       4             IoAcquireCancelSpinLock
   23    22 F09CF714   2 .rdata      1C             IoCreateDevice
   24    23 F09CF710   2 .rdata       4             IoDeleteDevice
   25    24 F09CF6F4   2 .rdata       8             IofCompleteRequest
   26    25 F09CF6F8   2 .rdata       4             IoReleaseCancelSpinLock
   27    26 F09CF700   2 .rdata       8             IoStartNextPacket
   28    27 F09CF6EC   2 .rdata      10             IoStartPacket
   29    28 F09CF728   2 .rdata       4             KeCancelTimer
   30    29 F09CF718   2 .rdata       C             KeInitializeDpc
   31    30 F09CF720   2 .rdata       C             KeInitializeEvent
   32    31 F09CF71C   2 .rdata       4             KeInitializeTimer
   33    32 F09CF6E4   2 .rdata       4             KeRemoveDeviceQueue
   34    33 F09CF6DC   2 .rdata       8             KeRemoveEntryDeviceQueue
   35    34 F09CF704   2 .rdata      10             KeSetTimer
   36    35 F09CF6CC   2 .rdata       4             KfLowerIrql
   37    36 F09CF6CB   2 .rdata       4             KfRaiseIrql
```

продолжение

Пример 1.12 (*продолжение*)

```
    38      37 F09CF6F0   2 .rdata        4          MmLockPagableDataSection
    39      38 F09CF6FC   2 .rdata        4          MmUnlockPagableImageSection
    40      39 F09CF72C   2 .rdata                   ntoskrnl_NULL_THUNK_DATA
    41      40 F09CF6D8   2 .rdata        8          RtlInitUnicodeString

-------------------------------------------------------------------

    13 non-NULL symbols
     0 exported symbols
```

Обратите внимание: идентификатор _allmul появился в списке дважды. Первый раз — это шлюз импорта для функции _allmul(), экспортируемой ntoskrnl.exe; второй раз — это просто перенаправитель вызова функции, который проходит через этот шлюз. Если добавить к команде ключ +d, чтобы увидеть декорированные идентификаторы, будет видно, что шлюз импорта _allmul на самом деле называется __imp___allmul, в то время как первоначальное имя перенаправителя — __allmul. Ясно, что декорированные имена действительно выполняют весьма важную задачу, несмотря на то что иногда они отвлекают внимание.

В этой главе был рассмотрен весьма объемный материал. Может быть, вы не ожидали, что так много можно сказать об отладчиках, отладочных интерфейсах API и файлах идентификаторов Windows 2000. Большинство книг по программированию не уделяют много внимания этой информации. Несмотря на это, я полагаю, что эти важные базовые знания помогут вам в создании собственных утилит отладки.

2 Интерфейс Native API Windows 2000

Основной материал этой вводной главы, посвященной интерфейсу Native API Windows 2000, в основном описывает взаимосвязи модулей операционной системы, образующих среду для работы этого базового программного интерфейса. Акцент сделан на описании центрального механизма, работающего через шлюз прерываний, при помощи которого Windows 2000 реализует передачу вызовов системных функций ядра из режима пользователя в режим ядра и возврат результатов обратно в пользовательский режим. Кроме того, будут описаны интерфейс Win32K и основные библиотеки времени выполнения, связанные с интерфейсом Native API, а также распространенные типы данных. В конце главы приводятся советы для тех разработчиков, которым требуется создавать приложения, взаимодействующие с Native API через библиотеку ntdll.dll.

Подробное описание архитектуры Windows 2000 можно найти в других источниках. Многое, написанное про Windows NT, верно также и для Windows 2000, поэтому в качестве хорошей вводной книги подойдут оба издания «Inside Windows NT», как и следующая за ними книга «Inside Windows 2000».

Наборы функций NT*() и ZW*()

Одна из самых интересных особенностей архитектуры Windows 2000 — это ее способность эмулировать различные операционные системы. Три встроенные подсистемы Windows 2000 обеспечивают работу приложений Win32, POSIX и OS/2. Наиболее широко распространена, безусловно, подсистема Win32, которая зачастую рассматривается разработчиками приложений как *сама* операционная система. Это неверное толкование нельзя ставить им в вину, поскольку такая точка зрения верна в случае устаревших операционных систем, таких как Windows 95 или 98, в которых реализация интерфейса Win32 действительно образует фундаментальную часть системы. Однако Windows 2000 спроектирована совершенно иначе. Хотя в подсистему Win32 входит системный модуль, называющийся kernel32.dll,

на самом деле это не настоящее ядро операционной системы, а только один из базовых компонентов подсистемы Win32. Во многих книгах по программированию обсуждение разработки программ для Windows NT и 2000 ограничивается описанием работы с прикладным программным интерфейсом (Application Programming Interface, API) Win32 и ничего не говорится о том, что платформа NT предоставляет еще один базовый интерфейс, называемый *Native API* (естественный API).

С интерфейсом Native API уже знакомы разработчики функционирующих в режиме ядра драйверов устройств или драйверов файловой системы, поскольку модули режима ядра расположены на более низком уровне, чем модули подсистем. Однако для доступа к этому интерфейсу нет необходимости спускаться до уровня драйверов, даже обычные приложения Win32 могут в любое время вызывать низкоуровневые функции Native API. С технической стороны этому ничто не мешает, просто Microsoft не поддерживает данный способ разработки приложений. Поэтому наблюдается явный дефицит информации по этой теме и работа приложений Win32 с интерфейсом Native API не обсуждается ни в комплекте Windows Platform Software Developer Kit (SDK), ни в Windows 2000 Device Driver Kit (DDK). Эта работа была оставлена другим, и данная книга — еще одна часть мозаики.

Уровни «недокументированности»

Большая часть материала этой книги относится к так называемой недокументированной информации. В общем случае это означает, что данная информация не публикуется Microsoft. Есть, тем не менее, несколько градаций «недокументированности», поскольку о такой гигантской операционной системе, как Windows 2000, можно опубликовать очень большой объем информации. Моя система деления информации на категории выглядит следующим образом:

- *официально документированная:* информация доступна в одном из таких источников, как книги, статьи или комплекты разработчика (development kit) фирмы Microsoft. Наиболее известные из них — комплекты SDK, DDK и библиотека разработчика MSDN (MicroSoft Developer Network library);
- *полудокументированная:* хотя эта информация официально не документируется, ее можно получить из официально распространяемых Microsoft файлов. Например, многие функции и структуры Windows 2000 не упомянуты в документации SDK или DDK, но появляются в заголовочных файлах и примерах программ. Наиболее важными источниками полудокументированной информации о Windows 2000 являются входящие в состав комплекта DDK заголовочные файлы ntddk.h и ntdef.h;
- *недокументированная, но не скрытая:* такую информацию нельзя найти в официальной документации, и она ни в какой форме не включена в продукты для разработчиков, однако частично ее можно получить, воспользовавшись инструментами отладки. К этой категории относится вся символьная информа-

ция в исполняемых файлах или файлах идентификаторов. Хорошим примером могут служить команды Kernel Debugger !processfields и !threadfields, копирующие из памяти имена и смещения недокументированных структур EPROCESS и ETHREAD (см. главу 1);

- *полностью недокументированная:* некоторые биты информации так хорошо спрятаны Microsoft, что получить к ним доступ можно только путем восстановления алгоритма работы программы и логических умозаключений. Сюда относится как информация о многочисленных деталях реализации, не представляющая интереса ни для кого, кроме разработчиков Windows 2000, так и информация, которая может оказаться неоценимой для системных программистов, особенно для разработчиков отладочного программного обеспечения. Извлечение информации о внутреннем устройстве системы — очень трудный, но и чрезвычайно интересный процесс для любителей головоломок.

Материал этой книги, описывающий внутреннее устройство Windows 2000, может быть в равной степени отнесен ко второму, третьему и четвертому пунктам данной классификации, так что каждый сможет найти здесь что-нибудь для себя.

Диспетчер системных вызовов

Связь между интерфейсами API подсистемы Win32 и Native API проще всего объяснить, показав зависимости между ключевыми модулями Win32 и ядром Windows 2000. На рис. 2.1 взаимосвязи модулей показаны в виде стрелок, соединяющих прямоугольники (модули). Стрелка, указывающая от модуля A к модулю B, означает зависимость A от B, то есть модуль A вызывает функции из модуля B. Модули, соединенные двойными стрелками, зависят друг от друга. На рис. 2.1 базовые поставщики функций Win32 API представлены модулями user32.dll, advapi.dll, gdi32.dll, rpcrt4.dll и kernel32.dll. Конечно, в данный API вносят вклад и другие библиотеки DLL, такие как version.dll, shell32.dll и comctl32.dll, но я опустил их для большей ясности. Рисунок 2.1 иллюстрирует интересную особенность: все вызовы Win32 API в конечном итоге направляются к модулю ntdll.dll, который передает их ntoskernel.exe.

Модуль ntdll.dll — это компонент операционной системы, содержащий интерфейс Native API. Более точно, ntdll.dll — это внешняя часть Natiive API пользовательского режима, «настоящий» же интерфейс реализован в ntoskernel.exe. По имени файла уже можно предположить, что это и есть *ядро операционной системы NT* (NT Operating System Kernel). Когда драйверам режима ядра требуется вызвать системные функции операционной системы, большую часть времени они обращаются к этому модулю. Основная задача модуля ntdll.dll — предоставить приложениям режима пользователя, в том числе и библиотекам DLL подсистемы Win32, определенное подмножество функций ядра. На рис. 2.1 стрелка, ведущая от ntdll.dll к ntoskernel.exe, помечена INT 2Eh, показывая, что Windows 2000 использует шлюз прерываний для переключения уровня привилегий процессора из режима пользо-

вателя в режим ядра. С точки зрения программистов режима ядра, код пользовательского режима — это агрессивный, содержащий ошибки и опасный код, который необходимо держать подальше от функций ядра. Переключение уровня привилегий между режимами пользователя и ядра во время обращения к API является одним из способов решения этой проблемы, позволяя управлять ходом событий. Вызывающее приложение никогда не касается байтов ядра, ему разрешено только смотреть на них издали.

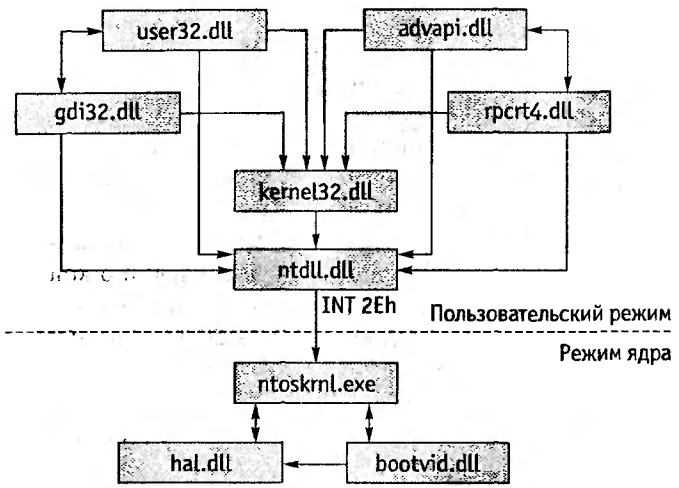

Рис. 2.1. Зависимости системных модулей

Например, функция Win32 API DeviceIoControl(), экспортируемая kernel32.dll, в конечном итоге вызывает экспортируемую ntdll.dll функцию NtDeviceIoControlFile(). При ее дизассемблировании открывается удивительно простая реализация, как видно из примера 2.1. Сначала в регистр процессора EAX загружается загадочное число 0x38, идентификатор вызова (dispatch ID). Затем в регистр EDX загружается адрес из области стека, значение которого равно текущему значению указателя стека ESP плюс четыре. Таким образом EDX будет показывать на адрес, расположенный сразу после записанного в стеке адреса возврата, сохраняемого непосредственно перед входом в функцию NtDeviceIoControlFile(). Конечно, это место временного хранения переданных функции аргументов. Следующая инструкция — просто INT 2Eh, реализующая переключение на обработчик прерываний, хранимый в записи 0x2E *таблицы дескрипторов прерываний* (Interrupt Descriptor Table, IDT). Выглядит знакомо, не правда ли? Фактически этот код во многом аналогичен старому вызову API INT 21h в DOS. Однако интерфейс INT 2Eh Windows 2000 играет гораздо большую роль, чем простой диспетчер вызовов API, — он выполняет функции главного шлюза между пользовательским режимом и режимом ядра. Обратите внимание на то, что данная реализация переключения режимов работает только для процессоров семейства Intel i386. На платформе Alpha для смены режимов применяются другие приемы.

Пример 2.1. Реализация ntdll.NtDeviceIoControlFile()

```
NtDeviceIoControlFile():
        mov     eax, 38h
        lea     edx, [esp+4]
        int     2Eh
        ret     28h
```

Интерфейс Native API Windows 2000 содержит 248 функций, реализованных подобным образом, что на 37 функций больше, чем в Windows NT 4.0. Эти функции легко узнать по префиксу имени функции Nt в списке функций, экспортируемых ntdll.dll. Однако ntdll.dll экспортирует 249 идентификаторов такого вида. Дело в том, что функция NtCurrentTeb() работает только в пользовательском режиме и поэтому ядру не передается. В табл. Б.1 приложения Б перечислены все функции Native API и их идентификаторы вызова по прерыванию INT 2Eh, если таковые имеются. В таблице также показано, какие функции экспортируются ntoskrnl.exe. Неожиданно, но из модулей режима ядра можно обращаться только к определенной части интерфейса Native API. С другой стороны, ntoskrnl.exe экспортирует два не предоставляемых ntdll.dll идентификатора Nt*, а именно NtBuildNumber и NtGlobalFlag, ни один из которых не связан с функцией. Это указатели на переменные ntoskrnl.exe, которые модуль драйвера может импортировать при помощи ключевого слова extern компилятора языка C. Таким способом ядро Windows 2000 экспортирует многие другие переменные, и в приведенном далее примере кода будет показано, как можно воспользоваться некоторыми из них.

Вы можете задаться вопросом, почему в табл. В.1 каждому модулю ntdll.dll и ntoskrnl.exe соответствует по два столбца, названных ntdll.Nt*, ntdll.Zw*, ntoskrnl.Nt* и ntoskrnl.Zw*. Дело в том, что оба модуля экспортируют два набора связанных идентификаторов интерфейса Native API. Один из них включает все имена с префиксом Nt, перечисленные в самом левом столбце таблицы. Другой набор содержит аналогичные имена, но префикс Nt заменен на Zw. Дизассемблирование ntdll.dll показывает, что каждой паре идентификаторов соответствует один и тот же код. Может показаться, что это напрасная трата памяти. Однако при дизассемблировании ntoskrnl.exe становится видно, что идентификаторы NT* указывают на настоящий код, а их разновидности Zw* ссылаются на заглушки INT 2Eh, код которых аналогичен коду примера 2.1. Это означает, что набор функций Zw* работает через шлюз, переключающий режимы пользователя и ядра, а функции NT* указывают непосредственно на код, выполняющийся после переключения режима.

В табл. В.1 следует отметить еще два момента. Во-первых, у функции NtCurrentTeb() нет соответствующего эквивалента Zw*. Это не очень большая проблема, поскольку экспортируемые ntdll.dll функции в любом случае одинаковы. Во-вторых, ntoskrnl.exe не всегда попарно экспортирует функции Nt/Zw. Для некоторых функций существует только одна из версий — Nt* или Zw*. Я не знаю, почему это так, предполагаю лишь, что ntoskrnl.exe экспортирует только функции, документированные в Windows 2000 DDK, а также функции, необходимые другим модулям операционной системы. Заметьте, что остальные функции Native API, тем не менее, все равно реализованы внутри ntoskrnl.exe. У них нет открытой точки входа, но, несомненно, к ним можно обратиться извне через шлюз INT 2Eh.

Таблицы дескрипторов системных вызовов

Из дизассемблированного кода в примере 2.1 видно, что прерывание INT 2Eh вызывается с двумя параметрами, передаваемыми через регистры процессора EAX и EDX. Я уже говорил, что загадочное число в EAX — это идентификатор вызываемой функции. Поскольку все вызовы Native API, за исключением NtCurrentTeb(), проходят через один и тот же вход, код обработчика прерывания INT 2Eh должен определить, какую именно функцию нужно вызвать. Для этого и передается идентификатор вызываемой функции. Для передачи вызова по окончательному назначению находящийся внутри модуля ntoskrnl.exe обработчик прерываний осуществляет поиск требуемой информации в таблице, используя значение в регистре EAX в качестве индекса. Эта таблица называется *таблицей системных вызовов* (System Service Table, SST), определение соответствующей структуры SYSTEM_SERVICE_TABLE языка C приведено в листинге 2.1. В нем также приведено определение структуры SERVICE_DESCRIPTOR_TABLE, членами[1] которой являются четыре таблицы SST. Первые две таблицы служат специальным целям.

Хотя обе таблицы являются фундаментальными типами данных, в Windows 2000 DDK они не документированы, откуда следует важное утверждение: многие напечатанные в книге фрагменты кода содержат недокументированные функции и типы данных. Следовательно, не гарантируется точная идентичность этой информации. Это верно для всех символьных идентификаторов, таких как имена структур, члены структур и параметры функций. При создании идентификаторов я старался использовать подходящие имена, руководствуясь схемой задания имен, применяемой в маленьком подмножестве известных идентификаторов (в том числе имен в файлах идентификаторов). Такой эвристический подход, вероятно, во многих случаях окажется неверным. Полную информацию содержит только оригинальный исходный код, но у меня нет к нему доступа. На самом деле я *не хочу* видеть исходный код, поскольку в этом случае мне потребуется заключить с Microsoft соглашение о конфиденциальности (Non-Disclosure Agreement, NDA), которое свяжет меня ограничениями, сильно мешающими при написании книги о недокументированной информации.

Давайте лучше вернемся к особенностям *таблицы дескрипторов системных вызовов* (Service descriptor Table, SDT). Из приведенного в листинге 2.1 определения видно, что две первые записи зарезервированы для модуля ntoskrnl.exe и части режима ядра подсистемы Win32, содержащейся внутри модуля win32k.sys. Передающиеся через таблицу SST модуля win32k вызовы порождаются модулями gdi32.dll и user32.dll. Модуль ntoskrnl.exe экспортирует указатель на свою главную таблицу SDT через идентификатор KeServiceDescriptorTable. Ядро поддерживает альтернативную таблицу SDT с именем KeServiceDescriptorTableShadow, но она не экспортируется. Обратиться к главной SDT из модуля режима ядра очень просто, для этого достаточно двух инструкций языка C, как показано в листинге 2.2. Первая инструкция — простое объявление переменной, предваряемое ключевым словом extern, которое говорит компоновщику, что переменная определена вне данного модуля и на этапе компоновки невозможно сопоставить ее идентификатору

[1] Здесь автор употребляет «array» в смысле абстрактного типа данных — совокупности элементов одинакового типа. В контексте языка C «array» по отношению к структуре выглядит не совсем корректно, поскольку в C есть базовый тип — массив. — *Примеч. перев.*

физический адрес. Все ссылки на этот идентификатор будут скомпонованы динамически, как только модуль будет загружен в адресное пространство процесса. Вторая инструкция в листинге 2.2 — как раз одна из таких ссылок. Присваивание KeServiceDescriptorTable переменной типа PSERVICE_DESCRIPTOR_TABLE приведет к установке динамической связи с ntoskrnl.exe, как при обращении к функции API, находящейся в модуле DLL.

Листинг 2.1. Структура таблицы дескрипторов системных вызовов (Service Descriptor Table, SDT)

```
typdef NTSTATUS (NTAPI *NTPROC) ();
typdef NTPROC *PNTPROC;
#define NTPROC_ sizeof (NTPROC)

//  -------------------------------------------------------
typedef struct _SYSTEM_SERVICE_TABLE
      {
      PNTPROC   ServiceTable;              // массив точек входа
      PDWORD    CounterTable;              // массив счетчиков
                                           // использования
      DWORD     ServiceLimit;              // число элементов
                                           // в таблице
      PBYTE     ArgumentTable;             // массив счетчиков
                                           // байт
      }
          SYSTEM_SERVICE_TABLE,
      * PSYSTEM_SERVICE_TABLE,
      **PPSYSTEM_SERVICE_TABLE;

//  -------------------------------------------------------

typedef struct _SERVICE_DESCRIPTOR_TABLE
      {
SYSTEM_SERVICE_TABLE  ntoskrnl;      // ntoskrnl.exe
                                     // (native API)
      SYSTEM_SERVICE_TABLE  win32k;  // win32k.sys
                                     // (поддержка gdi/user)
      SYSTEM_SERVICE_TABLE  Table3;  // не используется
      SYSTEM_SERVICE_TABLE  Table4;  // не используется
      }
          SERVICE_DESCRIPTOR_TABLE,
      * PSERVICE_DESCRIPTOR_TABLE,
      **PPSERVICE_DESCRIPTOR_TABLE;
```

Листинг 2.2. Обращение к таблице дескрипторов системных вызовов (Service Descriptor Table)

```
// импортировать указатель на SDT
extern PSERVICE_DESCRIPTOR_TABLE KeServiceDescriptorTable;

// создать ссылку на SDT
PSERVICE_DESCRIPTOR_TABLE psdt = KeServiceDescriptorTable;
```

Указатель ServiceTable каждой входящей в SDT структуры SST ссылается на массив[1] указателей на функции типа NTPROC, в котором удобно хранить функции интерфей-

[1] Надеюсь, читатели этой книги разберутся, что к чему, поэтому здесь и далее перевожу «array» как «массив». — *Примеч. перев.*

са Native API, аналогично типу данных PROC, используемому в программировании для Win32. Тип данных NTPROC определен вверху листинга 2.1. Функции интерфейса Native API, как правило, возвращают код NTSTATUS и используют соглашение о вызовах NTAPI, синоним __stdcall. В поле ServiceLimit содержится число элементов массива ServiceTable, которое в Windows 2000 по умолчанию равно 248. ArgumentTable — это массив элементов типа BYTE, каждый из которых соответствует записи в ServiceTable и показывает число байтов аргументов, доступных в стеке вызывающей функции. Эта информация вместе с указателем из регистра EDX необходима ядру, когда оно копирует аргументы из стека вызывающей функции в свой собственный стек, что будет описано ниже. Член структуры CounterTable не используется в свободной сборке (free build) Windows 2000. В отладочной сборке (debug build) он указывает на массив счетчиков использования каждой функции типа DWORD, эту информацию можно использовать для профилирования.

Опять же «array» употребляется не как базовый тип языка C, а как абстрактный тип данных. С точки зрения C типом данных ServiceTable является не массив указателей типа NTPROC, а просто указатель на NTPROC. Массивом было бы объявление NTPROC ServiceTable[]; хотя в C массив — это тоже указатель (то есть адрес), разница в том, что в случае массива он — константный. Впрочем, автор, видимо, рассматривает это с позиций ассемблера, а там — только условные типы данных, а массив представляется линейной областью памяти, для работы с которой нужен начальный адрес (имя массива). Тогда, конечно, нет особой разницы между, например, char ** x; и char * x[];.

Отладчик Kernel Debugger Windows 2000 позволяет легко вывести содержимое таблицы SDT. Если вы еще не установили это чрезвычайно важное приложение, обратитесь, пожалуйста, к главе 1. В примере 2.2 сначала я выполнил команду dd KeServiceDescriptorTable. Отладчик преобразовал этот символьный идентификатор в адрес 0x8046AB80 и вывел на экран шестнадцатеричный снимок памяти следующих 32-битных слов (DWORD), начиная с этого адреса. Имеют значение только первые четыре строки, соответствующие первым четырем членам структуры SDT из листинга 2.1. Для удобочитаемости эти строки выделены жирным шрифтом. Если приглядеться повнимательнее, можно заметить, что пятая строка выглядит точно так же, как первая — может быть, это еще одна таблица SDT? Появилась прекрасная возможность протестировать команду ln (List Nearest Symbols) отладчика Kernel Debugger. В примере 2.2 сразу после шестнадцатеричного снимка памяти KeServiceDescriptorTable я ввел команду ln 8046abc0. Очевидно, отладчику прекрасно известен адрес 0x8046abc0, и он преобразовывает его в символьный идентификатор KeServiceDescriptorTableShadow. Следовательно, подтверждается факт, что это и в самом деле вторая таблица SDT, которую поддерживает ядро. Таблицы явно различаются: последняя содержит записи для win32k.sys, в то время как в первой они отсутствуют. В обеих таблицах члены структуры Table3 и Table4 пусты. Для их заполнения модуль ntoskrnl.exe предоставляет удобную функцию API KeAddSystemServiceTable().

Пример 2.2. Исследование таблиц дескрипторов системных вызовов (Service Descriptor Table)

```
kd> dd KeServiceDescriptorTable
dd KeServiceDescriptorTable
8046ab80 804704d8 00000000 000000f8 804708bc
8046ab90 00000000 00000000 00000000 00000000
8046aba0 00000000 00000000 00000000 00000000
8046abb0 00000000 00000000 00000000 00000000
8046abc0 804704d8 00000000 000000f8 804708bc
8046abd0 a01859f0 00000000 0000027f a0186670
8046abe0 00000000 00000000 00000000 00000000
8046abf0 00000000 00000000 00000000 00000000
kd> ln 8046abc0
ln 8046abc0
(8047b3a0)       ntoskrnl!KeServiceDescriptorTableShadow

kd> ln 804704d8
ln 804704d8
(8046cd00)       ntoskrnl!KiServiceTable

kd> ln 804708bc
ln 804708bc
(8046d0e4)       ntoskrnl!KiArgumentTable

kd> ln a01859f0
ln a01859f0
(a016d8c0)       win32k!W32pServiceTable

kd> ln a0186670
ln a0186670
(a016e544)       win32k!W32pArgumentTable

kd> dd KiServiceTable
dd KiServiceTable
804704d8 804ab3bf 804ae86b 804bdef3 8050b034
804704e8 804c11f4 80459214 8050c2ff 8050c33f
804704f8 804b581c 80508874 8049860a 804fc7e2
80470508 804955f7 8049c8a6 80448472 804a8d50
80470518 804b6bfb 804f0cef 804fcb95 8040189a
80470528 804d06cb 80418f66 804f69d4 8049e0cc
80470538 8044c422 80496f58 804ab849 804aa9da
80470548 80465250 804f4bd5 8049bc80 804ca7a5

kd> dd KiArgumentTable
dd KiArgumentTable
804708bc 18 20 2c 2c 40 2c 40 44-0c 18 18 08 04 04 0c 10  .,,@,@D........
804708cc 18 08 08 0c 08 08 04 04-04 0c 04 20 08 0c 14 0c  ........... ....
804708dc 2c 10 0c 1c 20 10 38 10-14 20 24 1c 14 10 20 10  ,... .8.. $... .
804708ec 34 14 08 04 04 04 0c 08-28 04 1c 18 18 18 08 18  4.......(.......
804708fc 0c 08 0c 04 10 00 0c 10-28 08 08 10 00 1c 04 08  ........(.......
8047090c 0c 04 10 00 08 04 08 0c-28 10 04 0c 0c 28 24 28  ........(....($(
8047091c 30 0c 0c 0c 18 0c 0c 0c-0c 30 10 0c 0c 0c 0c 10  0........0......
8047092c 10 0c 0c 14 0c 14 18 14-08 14 08 08 04 2c 1c 24  .............,.$

kd> ln 8044c422
ln 8044c422
(80449c90)       ntoskrnl!NtClose
```

Заметьте, что я сократил выходные строки команды ln, оставив только существенную информацию.

По адресу 0x8046AB88 шестнадцатеричного снимка памяти KeServiceDescriptorTable, по которому должен быть расположен член структуры ServiceLimit, как и ожидалось, показано значение 0xF8 (248 в десятичной системе счисления). Значения ServiceTable и ArgumentTable представляют собой соответственно указатели на адреса 0x804704d8 и 0x804708bc. Это еще одна возможность применить команду ln, которая извлечет имена KiServiceTable и KiArgumentTable. Модуль ntoskrnl.exe не экспортирует ни один из этих символьных идентификаторов, но отладчик распознал их, обратившись к файлам идентификаторов Windows 2000. Команду ln также можно применить к указателям в таблице SST модуля win32k. Для членов структуры ServiceTable и ArgumentTable отладчик вернет соответственно имена W32pServiceTable и W32ArgumentTable, взятые из файла идентификаторов модуля win32k.sys. Если отладчик не сможет сопоставить адресам символьные идентификаторы, следует выполнить команду .reload, вследствие чего будут принудительно перезагружены все доступные файлы идентификаторов, и затем повторить попытку.

В оставшейся части примера 2.2 показаны первые 128 байт KiServiceTable и KiArgumentTable в шестнадцатеричной системе. Если верно то, что я до сих пор говорил про интерфейс Native API, то к функции NtClose() следует обращаться через 24-ю запись в таблице KiServiceTable, расположенную по адресу 0x80470538. Там находится значение 0x8044c422, выделенное жирным шрифтом в выходных данных команды dd KiServiceTable. Если применить команду ln к этому адресу, мы получим NtClose(). В качестве последнего теста давайте проверим 24-ю запись KiArgumentTable по адресу 0x804708d4. В Windows 2000 DDK говорится, что функция ZwClose() получает единственный аргумент типа HANDLE, поэтому общее число байт аргументов в стеке вызывающей функции должно равняться четырем. Неудивительно, что именно это значение было обнаружено в таблице аргументов. В данных, полученных после выполнения команды db KiArgumentTable, оно выделено жирным шрифтом.

Обработчик системных вызовов INT 2EH

Расположенный на стороне режима ядра шлюза INT 2Eh обработчик прерываний называется KiSystemService(). Опять же этот внутренний символьный идентификатор не экспортируется модулем ntoskrnl.exe, однако содержится в файлах идентификаторов Windows 2000. Следовательно, Kernel Debugger сможет найти его без особых затруднений. Обработчик KiSystemService() работает по следующей схеме:

1. Получает указатель на таблицу SDT от контрольного блока текущего потока.
2. Определяет требуемую таблицу SST в таблице дескрипторов системных вызовов (SDT), проверяя 12-й и 13-й биты идентификатора вызова (dispatch ID) в регистре EAX и выбирая соответствующий член структуры SDT. Идентификаторам из диапазона 0x0000-0x0FFF соответствует таблица ntoskrnl.exe, диапазон 0x1000-0x1FFF выделен в таблице win32k. Оставшиеся диапазоны 0x2000-0x2FFF и 0x3000-0x3FFF зарезервированы для дополнительных элементов SDT — Table3 и Table4. Если идентификатор превышает значение 0x3FFF, перед определением системного вызова ненужные биты будут сброшены.

3. Сверяет с 0-го по 11-й биты идентификатора вызова в регистре EAX с элементом ServiceLimit выбранной таблицы SST. Если идентификатор выходит за границы диапазона, возвращается код ошибки STATUS_INVALID_SYSTEM_SERVICE. Если таблица SST не используется, этот элемент будет равен нулю и для всех возможных идентификаторов вызова будет возвращаться код ошибки.
4. Сверяет указатель на стек аргументов в регистре EDX со значением MmUserProbeAddress. Это открытая переменная, экспортируемая из ntoskrnl.exe, которая обычно преобразуется в адрес 0x7FFF0000. Если указатель на стек аргументов не находится ниже этого адреса, возвращается STATUS_ACCESS_ VIOLATION.
5. Получает значение количества байт в стеке аргументов из ArgumentTable таблицы SST и копирует все аргументы функции из стека вызывающей функции в текущий стек режима ядра.
6. Получает указатель на системную функцию из ServiceTable таблицы SST и вызывает эту функцию.
7. Передает управление внутренней функции KiServiceExit() после возврата из системного вызова.

Интересно отметить, что обработчик INT 2Eh использует не глобальную таблицу SDT, доступную через указатель KeServiceDescriptorTable, а таблицу, адресуемую указателем из конкретного потока. Понятно, что с потоками могут быть связаны различные таблицы SDT. При инициализации потока функция KeInitializeThread() по умолчанию записывает в управляющий блок потока указатель KeServiceDescriptorTable. Эту настройку впоследствии можно изменить, задав, например, указатель KeServiceDescriptorTableShadow.

Интерфейс подсистемы Win32 режима ядра

На основе обсуждения таблицы SDT в предыдущем разделе можно сделать вывод, что наряду с Native API существует и второй основной интерфейс режима ядра, соединяющий модули Graphical Device Interface (GDI) и Window Manager (USER) подсистемы Win32 с компонентом режима ядра Win32K, находящимся в файле win32k.sys. Этот компонент был добавлен в Windows NT 4.0 для того, чтобы преодолеть присущее системе вывода на экран, входящей в Win32, ограничение производительности, вызванное первоначальной архитектурой подсистем в Windows NT. В Windows NT 3.x подсистема Win32 и ядро работали в рамках модели клиент-сервер, что вызывало постоянные переключения из режима пользователя в режим ядра и обратно. Значительные части системы вывода на экран были перенесены в модуль режима ядра win32k.sys, что существенно уменьшило эти накладные расходы.

Идентификаторы вызова Win32K

С появлением win32k.sys самое время обновить рис. 2.1. Рисунок 2.2 аналогичен исходному, добавился только прямоугольник win32.sys слева от ntoskrnl.exe. Я также провел стрелки от gdi32.dll и user32.dll к win32k.sys. Конечно, это не на 100% верно,

поскольку вызовы INT 2Eh из этих модулей в действительности направляются модулю ntoskrnl.exe, которому принадлежит обработчик прерываний. Тем не менее в конечном итоге эти вызовы обрабатываются в модуле win32k.sys, что и подразумевают стрелки.

Как было отмечено выше, интерфейс Win32K основан на диспетчере INT 2Eh во многом аналогично Native API. Единственное отличие в том, что Win32K использует идентификаторы вызова (dispatch ID) из другого диапазона. Хотя идентификаторы вызова всех функций интерфейса Native API лежат в диапазоне от 0x0000 до 0x0FFF, идентификаторы вызова Win32K принадлежат диапазону от 0x1000 до 0x1FFF. Как видно из рис. 2.2, первичные клиенты Win32K — модули gdi32.dll и user32.dll. Следовательно, при дизассемблировании этих модулей можно будет получить символьные имена, соответствующие идентификаторам вызова Win32K. Как становится ясно, только у небольшой части вызовов INT 2Eh существуют открытые имена в секциях экспортируемых функций, поэтому снова настало время для сеанса Kernel Debugger. Пример 2.3 демонстрирует выполнение команды dd W32pServiceTable. Для гарантии доступности идентификаторов win32k.sys предварительно выполнена команда .reload.

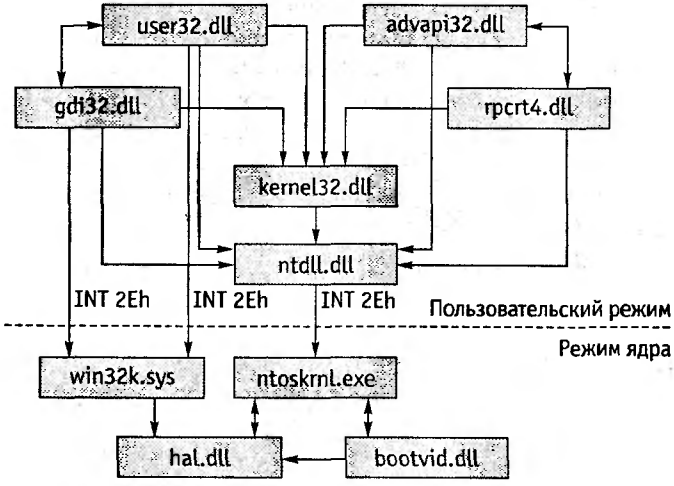

Рис. 2.2. Зависимости системных модулей, включая win32k.sys

Пример 2.3. Исследование системных вызовов Win32K

```
kd> .reload
.reload
Loading Kernel Symbols...
Unable to read image header for fdc.sys at f0798000 - status c0000001
Unable to read image header for ATMFD.DLL at beaaf000 - status c0000001
Loading User Symbols...
Unable to read selector for PCR for Processor 0
PPEB is NULL (Addr= 0000018c)
```

```
kd> dd W32pServiceTable
W32pServiceTable
a01859f0 a01077f0 a011f59e a000788a a01141e1
a0185a00 a0121264 a0107e05 a01084df a010520b
a0185a10 a0120a6f a008c9eb a00befa2 a007cb5c
a0185a20 a0085c9b a001e4e7 a0120fd1 a0122d19
a0185a30 a0085d0c a0122e73 a0027671 a006d1f0
a0185a40 a0043fe0 a009baeb a007eb9b a009eb05
a0185a50 a0043392 a007c14f a01229cc a0027470
a01B5a60 a001ad09 a00af751 a004e9f5 a004ef53

kd> ln a01077f0
a01077f0
(a00b316e)       win32k!NtGdiAbortDoc    |    (a00ba173)          win32k!IsRectEmpty
```

Последние три строки примера 2.3 демонстрируют применение команды ln к первой записи в шестнадцатеричном снимке памяти W32pServiceTable, из них сразу видно, что функция Win32K с нулевым идентификатором вызова называется NtGdiAbortDoc(). Эту процедуру можно повторить для всех 639 идентификаторов вызова, но процесс подстановки символьного идентификатора лучше автоматизировать. Я уже проделал это для вас, результаты собраны в приложении Б, табл. Б.2. Преобразование идентификаторов модулей gdi32.dll и user32.dll в идентификаторы win32k.sys происходит весьма просто: к идентификатору модуля GDI добавляется префикс NtGdi, а к идентификатору модуля USER — NtUser. Существуют, правда, незначительные исключения. Например, если идентификатор GDI начинается с символов Gdi, префикс сокращается до Nt, вероятно, чтобы избежать последовательности символов NtGdiGdi. В других случаях изменяется регистр символов (например, пара функций EnableEDUC() и NtGdiEnableEduc()) или теряется замыкающая буква W у функций для работы с Unicode (например, CopyAcceleratorTableW() и NtUserCopyAcceleratorTable()).

Подробное документирование всего интерфейса Win32K API — исключительно трудоемкая задача. Набор функций почти в три раза больше, чем у интерфейса Native API. Возможно, кто-нибудь когда-нибудь соберет вместе все разрозненные части и напишет исчерпывающее справочное руководство, как это сделал Гарри Неббет (Gary Nebbett) для Native API (Nebbett 2000), но для этой книги приведенной выше информации должно быть достаточно.

Библиотека времени выполнения Windows 2000

Функции Nt*() и Zw*(), образующие интерфейс Native API, — это важнейшая, но все-таки меньшая часть кода библиотеки ntdll.dll, которая экспортирует не менее 1179 идентификаторов. Из них 249 и 248 принадлежат соответственно наборам Nt*() и Zw*(), и все еще остается 682 функции, не проходящие через шлюз INT 2Eh. Ясно, что эта большая группа функций не связана с ядром Windows 2000. В чем же их предназначение?

Библиотека времени выполнения C

Если посмотреть на идентификаторы в разделе экспортируемых функций библиотеки ntdll.dll, обнаружится много имен функций в нижнем регистре, которые сразу узнает любой программист на C. Эти всем известные имена, такие как memcpy(), sprintf() и qsort(), входят в библиотеку времени выполнения C (C Runtime Library), включенную в ntdll.dll. Это верно и для модуля ntoskrnl.exe, который обладает похожим набором функций библиотеки C, хотя эти наборы не идентичны. В табл. Б.3 приложения Б перечислены все функции обоих наборов с информацией о том, к какому модулю принадлежит каждая функция.

Чтобы иметь возможность компоновки этих функций, достаточно добавить файл ntdll.lib из комплекта Windows 2000 DDK в список импортируемых библиотек, просматриваемых компоновщиком во время сопоставления идентификаторов. Если вы предпочитаете работать с диалоговыми окнами, выберите пункт Settings... в меню Project в среде разработки Visual C/C++, перейдите на вкладку Link, выберите категорию General и добавьте ntdll.dll в список Object/library modules. Результат будет такой же, если добавить в исходный код строку #pragma comment (linker, */defaultlib:ntdll.lib*). Последний вариант обладает тем преимуществом, что другие разработчики смогут перекомпоновать (rebuild) ваш проект с настройками Visual C/C++ по умолчанию.

Дизассемблирование кода некоторых функций библиотеки времени выполнения C, присутствующих в обоих модулях ntdll.dll и ntoskrnl.exe, показывает, что в этом случае модуль ntdll.dll не зависит от ntoskrnl.exe, как это было для функций Native API. Напротив, каждый модуль поддерживает собственную реализацию функций. Это верно и для всех остальных функций, представленных в этом разделе. Обратите внимание на то, что не все функции импортируются по имени. Например, если в драйвере режима ядра вы используете операции побитового сдвига >> и << для 64-разрядных чисел типа LARGE_INTEGER, компилятор и компоновщик автоматически импортируют из ntoskrnl.exe соответствующие функции _allshr() и _allshl().

Расширенная библиотека времени выполнения

Помимо стандартной библиотеки C Windows 2000 дополнительно предоставляет расширенный набор функций времени выполнения. Как и в предыдущем случае, ntdll.dll и ntoskrnl.exe реализуют их по отдельности, и опять же наборы реализованных функций перекрываются, но не совпадают полностью. Имена функций этой группы обладают общим префиксом Rtl (означающим Runtime Library). В табл. Б.4 приложения Б перечислены все эти функции по той же схеме, что и в табл. Б.3. В библиотеку времени выполнения Windows 2000 входят вспомогательные функции для часто встречающихся задач, выходящих за рамки возможностей библиотеки C. Например, одни функции связаны с вопросами безопасности, другие предназначены для работы со структурами данных, специфичными для Windows 2000, или же для управления памятью. Трудно понять, почему в комплекте Windows 2000 DDK Microsoft документирует всего 115 из 406 этих исключительно полезных функций.

Эмулятор операций с плавающей точкой

Я включу в эту коллекцию функций API еще один набор функций, предоставляемый ntdll.dll, просто чтобы показать, как много интересных функций сокрыто в этой сокровищнице. Перечисленные в табл. 2.1 имена должны показаться знакомыми программистам на языке ассемблера. Если убрать у имени функции префикс __e, получится мнемокоманда ассемблера устройства для выполнения операций с плавающей точкой (floating-point unit, FPU), встроенного в процессоры семейства i386. Фактически ntdll.dll включает в себя полнофункциональный эмулятор операций с плавающей точкой, представленный функциями, перечисленными в табл. 2.1. Это лишнее доказательство того, что в этой DLL содержится неисчерпаемое количество кода, который так и просится, чтобы его дизассемблировал исследователь архитектуры операционных систем.

Таблица 2.1. Интерфейс эмулятора операций с плавающей точкой модуля ntdll.dll

Имена функций			
__eCommonExceptions	__eFFREE	__eFLD64	__eFSTP32
__eEmulatorInit	__eFIADD16	__eFLD80	__eFSTP64
__eF2XM1	__eFIADD32	__eFLDCW	__eFSTP80
__eFABS	__eFICOM16	__eFLDENV	__eFSTSW
__eFADD32	__eFICOM32	__eFLDL2E	__eFSUB32
__eFADD64	__eFICOMP16	__eFLDLN2	__eFSUB64
__eFADDPreg	__eFICOMP32	__eFLDPI	__eFSUBPreg
__eFADDreg	__eFIDIV16	__eFLDZ	__eFSUBR32
__eFADDtop	__eFIDIV32	__eFMUL32	__eFSUBR64
__eFCHS	__eFIDIVR16	__eFMUL64	__eFSUBRPreg
__eFCOM	__eFIDIVR32	__eFMULPreg	__eFSUBRreg
__eFCOM32	__eFILD16	__eFMULreg	__eFSUBRtop
__eFCOM64	__eFILD32	__eFMULtop	__eFSUBreg
__eFCOMP	__eFILD64	__eFPATAN	__eFSUBtop
__eFCOMP32	__eFIMUL16	__eFPREM	__eFTST
__eFCOMP64	__eFIMUL32	__eFPREM1	__eFUCOM
__eFCOMPP	__eFINCSTP	__eFPTAN	__eFUCOMP
__eFCOS	__eFINIT	__eFRNDINT	__eFUCOMPP
__eFDECSTP	__eFIST16	__eFRSTOR	__eFXAM
__eFDIV32	__eFIST32	__eFSAVE	__eFXCH
__eFDIV64	__eFISTP16	__eFSCALE	__eFXTRACT
__eFDIVPreg	__eFISTP32	__eFSIN	__eFYL2X
__eFDIVR32	__eFISTP64	__eFSQRT	__eFYL2XP1
__eFDIVR64	__eFISUB16	__eFST	__eGetStatusWord
__eFDIVRPreg	__eFISUB32	__eFST32	NPXEMULATORTABLE
__eFDIVRreg	__eFISUBR16	__eFST64	RestoreEm87Context
__eFDIVRtop	__eFISUBR32	__eFSTCW	SaveEm87Context
__eFDIVreg	__eFLD1	__eFSTENV	
__eFDIVtop	__eFLD32	__eFSTP	

Дополнительную информацию о наборе инструкций с плавающей точкой можно получить из оригинальной документации для процессоров Intel 80386 и выше. Например, руководство по процессору Pentium можно загрузить в формате PDF с web-узла Intel по адресу http://developer.intel.com/design/pentium/manuals/. Руководство по инструкциям машинного языка называется «Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference» (Intel, 1999b). А также см. замечательные книги В. Юрова «Ассемблер. Учебник» и «Ассемблер. Справочник», выпущенные издательством «Питер».

Другие категории функций API

Помимо явно перечисленных в приложении Б и табл. 2.1 функций модули ntdll.dll и ntoskrnl.exe экспортируют много других функций, связанных с разнообразными компонентами ядра. Я не стал включать в книгу дополнительные длинные таблицы, а добавил только одну короткую, в которой перечислены возможные префиксы имен функций и сопоставленные им категории (табл. 2.2). Если модуль не экспортирует функции данной категории, в соответствующем столбце стоит запись «отсутствует».

Таблица 2.2. Префиксы категорий функций

Префикс	ntdll.dll	ntoskrnl.exe	Категория
__e		отсутствует	Эмулятор операций с плавающей точкой
Cc	отсутствует		Диспетчер кэша
Csr		отсутствует	Библиотека времени выполнения модели клиент-сервер
Dbg			Поддержка отладки
Ex	отсутствует		Поддержка исполняющей системы (executive)
FsRtl	отсутствует		Библиотека времени выполнения файловой системы
Hal	отсутствует		Диспетчер уровня аппаратных абстракций (Hardware Abstraction Layer, HAL)
Inbv	отсутствует		Драйвер инициализации системы/загрузки VGA (bootvid.dll)
Init	отсутствует		Инициализация системы
Interlocked	отсутствует		Потокобезопасное оперирование переменными
Io	отсутствует		Диспетчер ввода-вывода
Kd	отсутствует		Поддержка Kernel Debugger
Ke	отсутствует		Подпрограммы ядра
Ki			Обработка прерываний в ядре
Ldr			Загрузчик образа
Lpc	отсутствует		Функциональность локального вызова процедур (Local Procedure Call, LPC)
Lsa	отсутствует		Local Security Authority (LSA)
Mm	отсутствует		Диспетчер памяти
Nls			Поддержка национальных языков (National Language Support, NLS)

Префикс	ntdll.dll	ntoskrnl.exe	Категория
Nt			NT Native API
Ob	отсутствует		Диспетчер объектов
Pfx			Обработка префиксов
Po	отсутствует		Диспетчер электропитания
Ps	отсутствует		Поддержка процессов
READ_REGISTER	отсутствует		Чтение по адресу, находящемуся в регистре
Rtl			Библиотека времени выполнения Windows 2000
Se	отсутствует		Управление безопасностью
WRITE_REGISTER	отсутствует		Запись по адресу, находящемуся в регистре
Zw			Альтернативный интерфейс Native API
<other>			Вспомогательные функции и библиотека времени выполнения C

Многие функции ядра используют унифицированную схему именования вида ПрефиксДействиеОбъект. Например, функция NtQueryInformationFile() принадлежит к интерфейсу Native API, поскольку обладает префиксом Nt, и, очевидно, выполняет действие QueryInformation над объектом File. Большинство (но не все) функций подчиняются этому правилу, поэтому, обычно, по имени функции легко догадаться, что она делает.

Распространенные типы данных

При написании программ, взаимодействующих с ядром Windows 2000, как в пользовательском режиме через ntdll.dll, так и в режиме ядра через ntoskrnl.exe, зачастую приходится иметь дело с несколькими базовыми типами данных, редко встречающимися в мире программирования для Win32. Многие из них постоянно появляются в этой книге. В следующем разделе кратко описываются наиболее часто используемые типы данных.

Целочисленные типы

Традиционно существует несколько различных вариантов целочисленных типов данных. Ни заголовочные файлы Win32 Platform SDK, ни документация SDK не придерживаются какой-то строгой собственной спецификации — в них смешаны базовые типы данных C/C++ и несколько производных типов. В табл. 2.3 перечислены распространенные целочисленные типы данных и показано их соответствие друг другу. В столбце «MASM» показаны имена типов ассемблера Microsoft Macro Assembler (MASM). В комплекте Win32 Platform SDK определены типы BYTE, WORD и DWORD как псевдонимы соответствующих базовых типов данных C/C++. В столбцах «Псевдоним 1» и «Псевдоним 2» приведены другие часто используемые псевдонимы. Например, WCHAR представляет базовый символьный тип Unicode. В последнем столбце, «Со знаком», перечислены стандартные псевдонимы для соответствующих типов данных со знаком. Важно иметь в виду, что символы ANSI

типа CHAR — величины со знаком, в то время как символы Unicode типа WCHAR — беззнаковые. Такая несогласованность может привести к неожиданным побочным эффектам при преобразовании этих типов к другим целочисленным значениям в арифметических или логических выражениях.

Тип данных TBYTE (трактуется как «10-byte») ассемблера MASM в последней строке табл. 2.3 — 80-битное число с плавающей точкой, используемое при операциях с высокой точностью в устройстве FPU. Компилятор Microsoft Visual C/C++ не предоставляет соответствующего базового типа данных для программ Win32 — 80-битный тип *long double* 16-разрядных компиляторов Microsoft теперь трактуется как *double*, то есть 64-разрядное число с 11-битной экспоненциальной частью и 52-битной мантиссой, что соответствует спецификации IEEE real*8. Обратите внимание на то, что тип данных TBYTE в MASM не имеет ничего общего с типом данных TBYTE в Win32 (трактуется как «text byte»). Последний является определенным для удобства макросом, представляющим типы CHAR или WCHAR в зависимости от наличия или отсутствия в исходном коде строки #define UNICODE.

Таблица 2.3. Эквивалентные целочисленные типы данных

Бит	MASM	Базовый тип	Псевдоним 1	Псевдоним 2	Со знаком
8	BYTE	unsigned char	UCHAR		CHAR
16	WORD	unsigned short	USHORT	WCHAR	SHORT
32	DWORD	unsigned long	ULONG		LONG
32	DWORD	unsigned int	UINT		INT
64	QWORD	unsigned _int64[1]	ULONGLONG	DWORDLONG	LONGLONG
80	TBYTE	отсутствует			

В комплекте Windows Device Driver Kit (DDK) псевдонимы согласованы лучше. В документации и заголовочных файлах постоянно встречаются имена типов в столбцах «Псевдоним 1» и «Со знаком». Как программист с большим опытом работы на ассемблере я привык употреблять типы данных MASM, поэтому в заголовочных файлах на сопровождающем книгу компакт-диске по большей части используются имена типов из столбца «MASM».

Поскольку оперировать 64-разрядными целыми в 32-разрядной среде весьма неудобно. Windows 2000, как правило, не использует встроенный тип __int64 и производные от него типы. Взамен в заголовочном файле комплекта DDK ntdef.h определена соответствующая комбинация объединения и структуры, позволяющая рассматривать 64-битную величину либо как два 32-битных поля, либо как единое 64-битное поле. В листинге 2.3 показаны определения типов данных LARGE_INTEGER и ULARGE_INTEGER, реализующих соответственно целое со знаком и без знака. Управление знаком определяется использованием типов LONGLONG/ULONGLONG для 64-битного элемента QuardPart или типов LONG/ULONG для 32-битного элемента HighPart.

[1] Этот тип является встроенным только в компиляторе MS VC++, в ANSI C/C++ — это не базовый тип. Там нет встроенных 64-разрядных вычислений, что, впрочем, и так все прекрасно знают. — *Примеч. перев.*

Строки

В программировании для Win32 для представления строк ANSI и Unicode широко используются основные типы PSTR и PWSTR, определенные соответственно как CHAR* и WCHAR* (см. табл. 2.3). Дополнительный псевдотип PTSTR эквивалентен одному из типов PWSTR или PSTR, что управляется соответственно наличием или отсутствием в исходном коде макроопределения #define UNICODE. Таким образом можно поддерживать версии приложения для стандартов ANSI и Unicode при помощи одного и того же кода. В основном эти строки представляют собой просто указатели на заканчивающиеся нулем массивы символов CHAR или WCHAR. При работе с ядром Windows 2000 приходится иметь дело с совершенно другим представлением строк. Наиболее распространен тип данных UNICODE_STRING, определение этой структуры из трех элементов приведено в листинге 2.4.

Листинг 2.3. Типы данных LARGE_INTEGER и ULARGE_INTEGER

```
typedef union _LARGE_INTEGER
      {
      struct
           {
           ULONG LowPart;
           LONG HighPart;
           };
      LONGLONG QuadPart;
      }
      LARGE_INTEGER, *PLARGE_INTEGER;

typedef union _ULARGE_INTEGER
      {
      struct
           {
           ULONG LowPart;
           ULONG HighPart;
           };
      ULONGLONG QuadPart;
      }
      ULARGE_INTEGER, *PULARGE_INTEGER;
```

Листинг 2.4. Строковые типы, определенные как структуры

```
typedef struct _UNICODE_STRING
      {
      USHORT Length;
      USHORT MaximumLength;
      PWSTR  Buffer;
      }
      UNICODE_STRING, *PUNICDDE_STRING;

typedef struct _STRING
      {
      USHORT Length;
      USHORT MaximumLength;
      PCHAR  Buffer;
      }
      STRING, *PSTRING;

typedef STRING ANSI_STRING, *PANSI_STRING;
typedef STRING OEM_STRING,  *POEM_STRING;
```

В поле структуры Length хранится теущая длина строки в байтах — но не в символах! Элемент Buffer указывает на блок памяти с данными строки, размер которого задан в MaximumLength, снова не в символах, а в байтах. Так как размер символов Unicode составляет 16 бит, значение в Length всегда равно двукратному числу символов строки. Как правило, строка, на которую указывает Buffer, завершается нулем. Однако некоторые модули режима ядра могут полностью полагаться в этом вопросе на значение Length и не заботиться о добавлении завершающего нуля[1], поэтому проявляйте осторожность в сомнительных случаях.

Вариант ANSI определяющей строку структуры в Windows 2000 называется просто STRING, как показано в листинге 2.4. Для удобства в файле ntdef.h также определены псевдонимы этой структуры ANSI_STRING и OEM_STRING, чтобы отделить друг от друга 8-битовые строки символов, принадлежащих к различным кодовым страницам (по умолчанию кодовая страница ANSI — 1252, кодовая страница OEM — 437). Тем не менее преобладающим строковым типом данных ядра Windows 2000 является UNICODE_STRING. Только изредка вы будете сталкиваться с 8-битовыми строками.

На рис. 2.3 приведены примеры двух типичных строк UNICODE_STRING. Расположенный слева пример состоит из двух независимых блоков памяти: структуры UNICODE_STRING и массива 16-битных символов Unicode типа PWCHAR. Это, вероятно, наиболее распространенный строковый тип, который можно найти в областях данных Windows 2000. Справа приведена часто встречающаяся схема представляющей особый случай строки, в которой и структура UNICODE_STRING, и массив символов PWCHAR расположены в одном и том же блоке памяти. Несколько функций ядра, в том числе некоторые функции из Native API, возвращают структурированную системную информацию в непрерывных блоках памяти. Если в этих данных содержатся строки, они зачастую хранятся в виде встроенных строк типа UNICODE_STRING, как показано в правой части рис. 2.3. Например, функция NtQuerySystemInformation() из кода примера в главе 1 постоянно пользуется этим специальным представлением строки.

Нет необходимости напрямую работать с этими структурами, поскольку модули ntdll.dll и ntoskrnl.exe экспортируют богатый набор функций API времени выполнения, таких как RtlCreatUnicodeString(), RtlInitUnicodeString(), RtlCopyUnicodeString() и им подобных. Эквивалентная функция обычно существует и для типов данных STRING и ANSI_STRING. Многие такие функции официально документированы в комплекте DDK, но для некоторых описание отсутствует. Несмотря на это, обычно легко догадаться, что делает оперирующая со строками недокументированная функция и какие аргументы она ожидает. Основное преимущество типа данных UNICODE_STRING и его родственников состоит в том, что в них явно указан размер содержащего строку буфера. Это очень удобно, например, когда строка UNICODE_STRING передается в качестве параметра в функцию, которая при работе с ее содержимым может увеличить длину строки. Чтобы определить, хватит ли для этого места в буфере строки, функции достаточно будет только свериться со значением MaximumLength.

[1] В оригинале: «zero character». Следует не забывать, что символ '0' совсем не то, что значение 0 ('0\'), завершающее строку. — *Примеч. перев.*

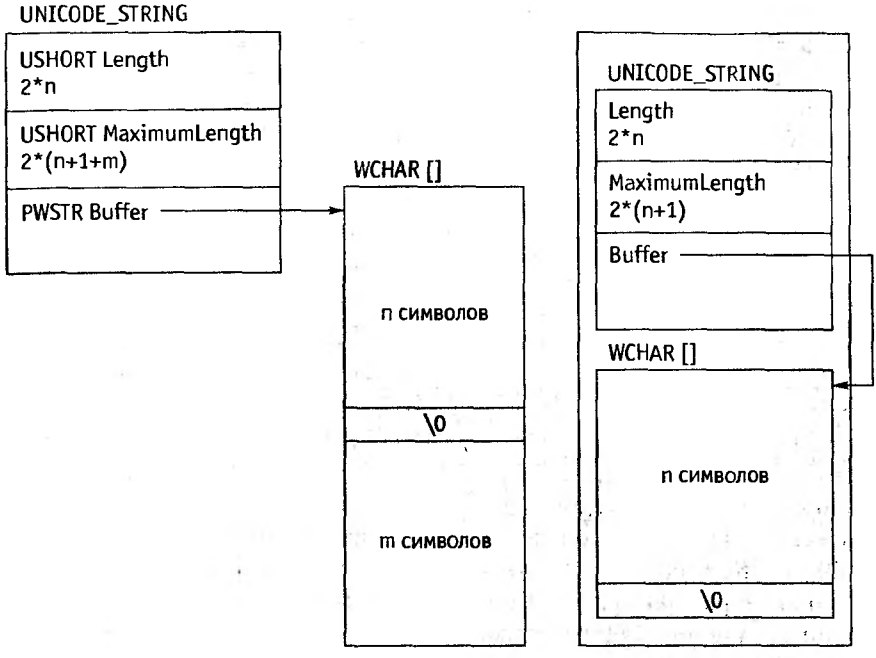

Рис. 2.3. Примеры строк типа UNICODE_STRING

Структуры

Для некоторых функций API ядра, работающих с объектами, требуется, чтобы объекты соответствовали правильно заполненной структуре OBJECT_ATTRIBUTES, определение которой приведено в листинге 2.5. Например, у функции NtOpenFile() нет параметра[1] с типом данных PWSTR или PUNICODE_STRING для пути к открываемому файлу, путь передается через элемент ObjectName структуры OBJECT_ATTRIBUTES. Установка значений полей этой структуры обычно не составляет труда. Помимо ObjectName необходимо указать также значения элементов Length и Attributes. Поле Length следует установить в значение sizeof (OBJECT_ATTRIBUTES), а в элементе Attributes должен содержаться ряд значений типа OBJ_* из файла ntdef.h, например OBJ_CASE_INSENSITIVE, если имя объекта следует рассматривать без учета регистра. Конечно, поле ObjectName — это указатель на UNICODE_STRING, а не простой указатель типа PWSTR. Остальные поля, если они не требуются, можно установить в NULL.

В то время как структура OBJECT_ATTRIBUTES подробно описывает входные данные функции API, структура IO_STATUS_BLOCK из листинга 2.6 предоставляет информацию о результате запрошенной операции. Эта структура весьма проста: элемент

[1] В оригинале: «arguments». Обычно под «параметрами» подразумеваются переменные в определении функции (формальные параметры), а под «аргументами» — фактически переданные в функцию значения (фактические параметры). Это, конечно, необязательное соглашение, но все же считаю необходимым обратить на это внимание читателя. — *Примеч. перев.*

Status содержит код состояния NTSTATUS, который может принимать либо значение STATUS_SUCCESS, либо значение, соответствующее одному из кодов ошибок, определенных в заголовочном файле DDK ntstatus.h. В случае успеха поле Information будет содержать дополнительные данные, смысл которых зависит от конкретной операции. Например, если функция возвращает блок данных, в этом поле, как правило, находится размер блока.

Структура LIST_ENTRY, описание которой приведено в листинге 2.7, — еще один повсеместно используемый в Windows 2000 тип данных. В ядре эта простая структура применяется для упорядочивания объектов в дважды связанных списках. Довольно часто один объект входит в несколько списков, вследствие чего в определении объекта присутствуют несколько структур LIST_ENTRY. Поле Flink (forward link, ссылка вперед) указывает на следующий элемент списка, а поле Blink (backward link, ссылка назад) — на предыдущий. Связи всегда указывают на другую структуру LIST_ENTRY, но не на сам объект, в который она включена. Обычно связанные списки замыкаются в круг, то есть последняя связь Flink указывает на первый элемент LIST_ENTRY в цепочке, а первая связь Blink указывает на конец списка. Такая структура данных позволяет легко проходить по связанному списку в обоих направлениях, начиная с любого его конца, или даже с одного из расположенных в середине элементов. Если программе нужно пройти по списку объектов, ей необходимо сохранить адрес начального элемента, чтобы знать, когда нужно остановиться. Если в списке только один элемент, структура LIST_ENTRY должна ссылаться на себя — то есть оба поля Flink и Slink должны указывать на свою собственную структуру LIST_ENTRY.

Листинг 2.5. Структура OBJECT_ATTRIBUTES

```
typedef struct _OBJECT_ATTRIBUTES
      {
      ULONG              Length;
      HANDLE             RootDirectory;
      PUNICODE_STRING    ObjectName;
      ULONG              Attributes;
      PVOID              SecurityDescriptor;
      PVOID              SecurityQualityOfService;
      }
      OBJECT_ATTRIBUTES,  *POBJECT_ATTRIBUTES;
```

Листинг 2.6. Структура IO_STATUS_BLOCK

```
typedef struct _IO_STATUS_BLOCK
      {
      NTSTATUS  Status;
      ULONG     Information;
      }
      IO_STATUS_BLOCK,  *PSTATUS_BLOCK;
```

Листинг 2.7. Структура LIST_ENTRY

```
typedef struct _LIST_ENTRY
      {
      struct    _LIST_ENTRY *Flink;
      struct    _LIST_ENTRY *Blink;
      {
      LIST_ENTRY,  *PLIST_ENTRY;
```

Рисунок 2.4 иллюстрирует взаимосвязи между элементами списка объектов. Объекты A1, A2 и A3 входят в список из трех элементов. Обратите внимание на то, что A3.Flink указывает обратно на A1, а A1.Blink указывает на A3. Объект B1 в правой части рисунка является единственным элементом вырожденного списка. Поэтому его поля Flink и Blink указывают на один и тот же адрес внутри объекта B1. Типичными примерами дважды связанных списков могут служить списки процессов и потоков. Внутренняя переменная PsActiveProcessHead, расположенная в секции .data модуля ntoskrnl.exe, является структурой типа LIST_ENTRY, которая представляет первый (а в зависимости от значения указателя Blink — также и последний) элемент списка системных процессов. В окне консоли отладчика Kernel Debugger можно пройти по этому списку, выполнив сначала команду dd PsActiveProcessHead, а затем при помощи операций копирования и вставки задавая данные последующих команд dd для значений Flink и Blink. Это, конечно, ужасно нудный способ изучения процессов Windows 2000, но он может помочь заглянуть внутрь базовой архитектуры системы. Интерфейс Native API Windows 2000 предлагает гораздо более удобные способы перечисления процессов, такие как функция NTQuerySystemInformation().

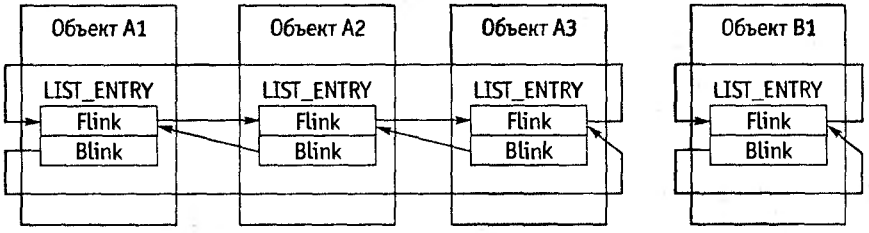

Рис. 2.4. Примеры дважды связанных списков

Функции API работают с процессами и потоками, такими как NtOpenProcess() и NtOpenThread(), одновременно оперируя потоком и процессом при помощи структуры CLIENT_ID, показанной в листинге 2.8. Элементы UniqueProcess и UniqueThread определены как переменные типа HANDLE (дескриптор), однако в буквальном смысле это не дескрипторы, а целочисленные идентификаторы процесса и потока. Это те же самые идентификаторы, которые возвращают стандартные функции Win32 API GetCurrentProcessId() и GetCurrentThreadId(), возвращающие значение типа DWORD.

Структура CLIENT_ID также используется модулем Executive Windows 2000 в качестве глобального идентификатора потока в системе. Например, если в отладчике Kernel Debugger вывести параметры текущего потока командой !thread, в первой выходной строке будет показан идентификатор CLIENT_ID этого потока в виде «Cid ppp.ttt.», где «ppp» — значение поля UniqueProcess, а «ttt» —значение поля идентификатора потока UniqueThread.

Листинг 2.8. Структура CLIENT_ID

```
typedef struct _CLIENT_ID
      {
      HANDLE UniqueProcess;
      HANDLE UniqueThread;
      }
      CLIENT_ID, *PCLIENT_ID;
```

Работа с Native API

Для драйверов режима ядра обращение к Native API — обычное дело, аналогично вызову функций Win32 API из приложения пользовательского режима. Все необходимое для обращения к функциям Native API, предоставляемым модулем ntoskrnl.exe, содержится в библиотечных и заголовочных файлах Windows 2000 DDK. С другой стороны, Win32 Platform SDK практически никак не поддерживает работу приложений с функциями Native API, экспортируемыми модулем ntdll.dll. «Практически», потому что один важный элемент все-таки включен: это импортируемая библиотека ntdll.lib, расположенная в каталоге \Program Files\Microsoft Platform SDK\Lib. Без этой библиотеки вызывать экспортируемые ntdll.dll функции было бы затруднительно.

Включение в проект импортируемой библиотеки ntdll.dll

Для успешной компиляции и компоновки кода пользовательского режима, использующего функции API библиотеки ntdll.dll, необходимо учесть следующих четыре факта:

1. Заголовочные файлы Platform SDK не содержат прототипов этих функций.
2. В файлах SDK отсутствуют некоторые базовые структуры данных, используемые этими функциями.
3. Заголовочные файлы SDK и DDK несовместимы — нельзя добавить в исходные файлы Win32 C строку #include <ntddk.h>.
4. Библиотека ntdll.dll по умолчанию не включена в список импортируемых библиотек Visual C/C++.

Последняя проблема легко решаема: достаточно исправить для приложения настройки проекта или добавить в исходный код строку #pragma comment (linker, "defaultlib:ntdll.lib"), как было объяснено ранее в этой главе в разделе «Библиотека времени выполнения Windows 2000». В результате применения этой директивы компоновщика на этапе компиляции в часть команды компоновки /defaultlib будет добавлена библиотека ntdll.lib. Проблема с отсутствующими определениями гораздо серьезнее. Так как в программах на простом C невозможно одновременно использовать заголовочные файлы SDK и DDK, наименее затратным решением будет создание собственного заголовочного файла, содержащего по крайней мере все определения, необходимые для вызова нужных функций API из ntdll.dll. К счастью, нет необходимости создавать все с нуля. Большая часть основной информации, которая может понадобиться, содержится в файле w2k_def.h в каталоге \src\common\include компакт-диска с примерами программ. Этот заголовочный файл будет играть важную роль в главах 6 и 7. Поскольку он совместим с проектами как режима пользователя, так и режима ядра, в коде пользовательского режима необходимо вставить строку #define _USER_MODE где-либо перед строкой #include <w2k_def.h>, чтобы включить определения DDK, отсутствующие в SDK.

Достаточно много о программировании для Native API было уже опубликовано. Три хороших источника с подробной информацией по этой теме приведены ниже в порядке публикации:

- Марк Руссинович (Mark Russinovich) опубликовал статью «Inside the Native API» на web-узле *sysinternals.com*, которую можно загрузить с http://www.sysinternals.com/ntdll.htm;
- в номере за ноябрь 1999 года журнала «Dr. Dobb's Journal» (DDJ) опубликована моя статья «Inside Windows NT System Data», в которой, помимо прочего, подробно описан способ взаимодействия с ntdll.dll и приводится много примеров программ, облегчающих эту задачу. Код примеров можно загрузить с web-узла *DDJ* по адресу http://www.ddj.com/ftp/1999/1999_11/ntinfo.zip. Обратите внимание на то, что эта статья посвящена только Windows NT 4.0;
- недавно опубликованная библия по Native API Гарри Неббета (Gary Nebbet), «Windows NT/2000 Native API Reference»[1]. В ней не очень много примеров кода, но эта книга полностью охватывает все функции Native API систем Windows NT 4.0 и Windows 2000, включая требуемые им структуры данных и другие определения. Эта книга идеально дополнит приведенные выше статьи.

Представленная в главе 6 библиотека примеров w2k_call.dll демонстрирует типичные способы применения файла w2k_def.h. В главе 6 также обсуждается альтернативный способ обращения к ядру Windows 2000 из пользовательского режима, который не ограничен набором функций интерфейса Native API. Фактически этот прием не ограничен только ntoskrnl.exe, он применим к *любому* модулю, загруженному в память ядра, который либо экспортирует функции API, либо поставляется с соответствующими файлами идентификаторов .dbg или .pdb. Как вы скоро увидите, в оставшихся главах книги вас ожидает еще много интересного. Но прежде, чем перейти к обсуждению этого материала, мы рассмотрим некоторые фундаментальные концепции и техники.

[1] А русскоязычному читателю порекомендуем выпущенную издательством «Питер» книгу Дана Эпплмана «Win32 API и Visual Basic». Несмотря на название, эта книга в первую очередь про Win32 API. От Visual Basic в ней только тексты примеров. — *Примеч. ред.*

3 Разработка драйверов режима ядра

В последующих главах нам часто придется иметь дело с системными ресурсами, которые доступны только для исполняемого кода, работающего в режиме ядра. Большая часть исходного кода, рассматриваемого далее в данной книге, имеет форму подпрограмм драйвера режима ядра. Таким образом, для того чтобы освоить дальнейший материал, вам потребуются базовые знания о том, как устроен драйвер ядра и как разработать собственный драйвер. Не думаю, что абсолютно все читатели обладают опытом разработки подобного низкоуровневого программного обеспечения, поэтому в данной главе я решил поместить краткое введение в разработку драйверов, работающих на уровне ядра. При создании драйверов я буду активно пользоваться специальным мастером генерации базового исходного кода драйверов. Этот мастер содержится на прилагаемом к книге компакт-диске.

В данной главе также содержатся базовые сведения о диспетчере управления службами Windows 2000 Service Control Manager, который позволяет загружать и выгружать драйверы прямо в процессе функционирования ОС, а также осуществляет управление ими. Благодаря использованию этого диспетчера для того, чтобы протестировать работу драйвера, вам не потребуется перезагружать ОС. Другими словами, на изменение исходного кода, компиляцию и тестирование уходит не так много времени. Заголовок главы многих может сбить с толку — термином «*драйвер*», как правило, обозначается низкоуровневая программа, осуществляющая управление некоторым аппаратным устройством. Очень многие программисты, работающие на уровне ядра, в основном занимаются разработкой именно таких программ. Однако многоярусная модель драйверов, используемая Windows 2000, позволяет использовать драйверы не только для управления оборудованием, но и для выполнения некоторых других задач. Драйверы, работающие на уровне ядра, могут использоваться для выполнения самых разнообразных функций и даже могут играть роль высокоуровневых библиотек DLL, работающих в пользовательском режиме. Такая библиотека доступна для высокоуровневых пользовательских приложений, однако ее функции выполняются на более высоком уровне привилегий процессора, кроме того, она использует иной программный интерфейс. В дан-

ной книге инфраструктура драйверов, встроенная в Windows 2000, не будет использоваться для управления аппаратным обеспечением компьютера. Вместо этого мы будем использовать драйверы для изучения внутренней структуры Windows 2000. Драйверы режима ядра будут использоваться нами в качестве космических челноков, которые осуществляют связь между маленьким миром пользовательского режима и необъятной вселенной режима ядра Windows 2000.

Создание скелета драйвера

Программист, в течение длительного времени занимавшийся разработкой приложений и библиотек Win32, может почувствовать себя совершенным новичком, когда он попробует написать свой первый драйвер режима ядра. Причина этого в том, что код режима ядра исполняется в совершенно иной операционной среде. Программист Win32 работает с системными компонентами, являющимися частью подсистемы Win32, входящей в состав Windows 2000. Некоторые другие программисты могут обращаться к другим системным компонентам, например к компонентам, входящим в состав подсистем POSIX или OS/2. Эти подсистемы также входят в состав Windows 2000. Благодаря концепции подсистем операционная система Windows 2000 может вести себя подобно хамелеону – она может эмулировать различные операционные системы, обеспечивая доступ к их программным интерфейсам при помощи различных добавляемых в нее подсистем. В отличие от высокоуровневых приложений модули, работающие в режиме ядра, располагаются ниже уровня подсистем, поэтому они обязаны обращаться к функциям более низкоуровневого интерфейса операционной системы. На этом уровне не существует каких-либо подсистем, поэтому исполняемый код режим ядра «видит» реальный интерфейс операционной системы Windows 2000, который можно рассматривать как последний рубеж, отделяющий разработанную вами программу от кода ядра ОС. На самом деле утверждение о том, что в режиме ядра отсутствуют какие-либо подсистемы, не совсем точно. В главе 2 уже упоминалось о том, что модуль win32k.sys доступен в режиме ядра и является частью Win32 GUI и диспетчера окон Windows Manager. Оба этих компонента работают в режиме ядра для обеспечения более высокого быстродействия. Однако из всех функций win32k.sys лишь небольшая часть доступна для программ пользовательского режима в виде функций Win32 API, входящих в состав gdi32.dll и user32.dll. Поэтому интерфейс Win32K можно рассматривать как опору, благодаря которой Win32 API опирается на фундамент режима ядра. Кроме того Win32K выполняет функции высокоскоростного ядра графической подсистемы.

Комплект разработки драйверов Windows 2000 Device Driver Kit

Программы, работающие в режиме ядра, не могут пользоваться обычными программными интерфейсами Win32, поэтому при разработке драйверов режима ядра нельзя использовать привычные для многих программистов Win32 заголовочные

файлы и библиотеки. Для разработки программ Win32 компания Microsoft предлагает использовать комплект разработки программ Platform Software Development Kit (SDK), однако для разработки драйверов режима ядра вам потребуется комплект Windows 2000 Driver Development Kit (DDK). Помимо документации в этот комплект входят специальные заголовочные файлы и библиотеки, необходимые для обеспечения взаимодействия разрабатываемого вами кода и других модулей ядра Windows 2000. После того как вы установите DDK, вы должны запустить Microsoft Visual C и добавить пути к файлам DDK к списку файлов, используемых компилятором и компоновщиком при создании исполняемого файла. Для этого в главном меню выберите Tools (сервис) ▸ Options (параметры) и перейдите на вкладку Directories (каталоги). В ниспадающем списке Show directories for: (показать каталоги для) выберите Include files (включаемые файлы) и добавьте в перечень каталогов каталог, содержащий включаемые файлы комплекта DDK. Эта процедура проиллюстрирована на рис. 3.1. По умолчанию DDK устанавливается в каталоге с именем \NTDDK, а включаемые файлы содержатся в подкаталоге с именем \NTDDK\inc. Добавив имя каталога в перечень, используйте стрелку вверх для того, чтобы переместить этот каталог в нужную позицию списка. В большинстве ситуаций указанный вами каталог включаемых файлов DDK должен располагаться на второй сверху позиции списка — сразу же после каталога включаемых файлов Platform SDK. Всегда располагайте каталоги с включаемыми файлами, изначально входящими в комплект Microsoft Visual Studio, в конце списка, так как многие из этих файлов заменяются более свежими версиями этих же файлов из комплектов SDK и DDK.

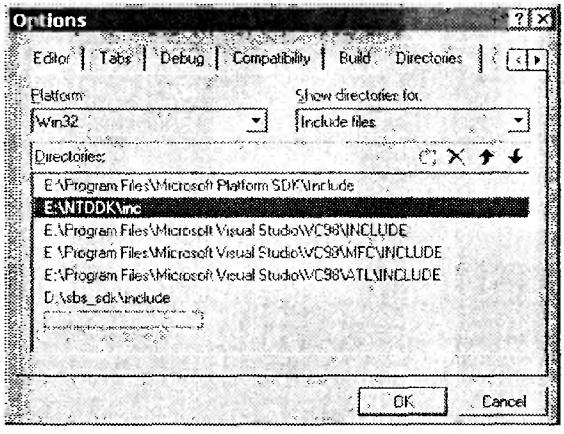

Рис. 3.1. Добавление в Visual Studio каталога заголовочных файлов DDK

После того как вы указали путь к каталогу заголовочных файлов DDK, сделайте то же самое в отношении импортируемых библиотек DDK. В комплект DDK входят две версии каждого библиотечного файла. Первая из них — это окончательная версия (release), готовая к использованию в конечном продукте. Вторая версия является отладочной (debug) — ее удобнее применять в процессе разработки драй-

вера для облегчения отладки. Первую версию иногда называют свободной (free), а вторую — проверенной (checked), согласно этим обозначениям именуются каталоги, в которых содержатся эти библиотеки: \NTDDK\libfre\i386 и \NTDDK\libchk\i386 соответственно. На рис. 3.2 проиллюстрирован процесс добавления каталога импортируемых библиотек в конфигурацию Visual Studio. Чтобы добавить каталог библиотек, в ниспадающем списке Show directories for: (показать каталоги для) выберите Library files (библиотечные файлы), укажите каталог библиотек и, используя стрелку вверх, переместите соответствующую этому каталогу запись в нужную позицию списка каталогов.

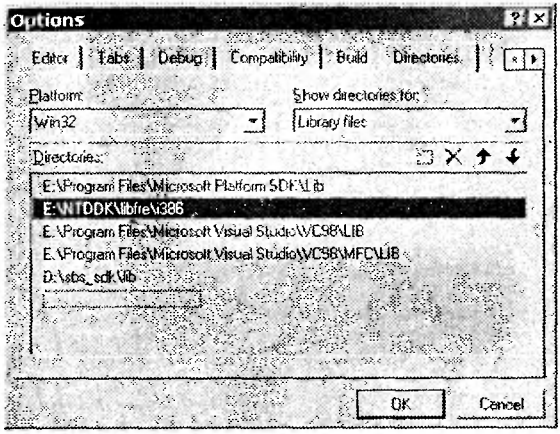

Рис. 3.2. Добавление в Visual Studio каталога импортируемых библиотек DDK

Среда программирования DDK несколько отличается от модели Win32. К наиболее значительным отличиям следует отнести:

- В исходный код программ Win32 обязательно следует включить заголовочный файл windows.h, однако при разработке драйвера режима ядра этот заголовочный файл использовать нельзя. Вместо него в текст драйвера следует включить файл ntddk.h.
- Функция, являющаяся основной точкой входа для драйвера, называется DriverEntry(), но не WinMain() и не main(), как это принято в обычных программах. Прототип функции DriverEntry() показан в листинге 3.1.
- Помните, что некоторые типы данных Win32, такие как BYTE, WORD и DWORD, недоступны при разработке драйверов режима ядра. Вместо них в среде DDK следует использовать UCHAR, USHORT, ULONG и т. п. Однако вы с легкостью можете самостоятельно определить привычные для вас типы, как это сделано в листинге 3.2.

Также, разрабатывая драйверы режима ядра, необходимо принимать во внимание три важных отличия между Windows NT 4.0 DDK и Windows 2000 DDK:

- Базовый каталог Windows NT 4.0 DDK по умолчанию называется \DDK, в то время как базовый каталог Windows 2000 DDK по умолчанию имеет имя \NTDDK.

- В Windows NT 4.0 DDK основной заголовочный файл ntddk.h располагается в базовом каталоге, в то время как в Windows 2000 DDK этот файл размещается в подкаталоге ddk базового каталога.
- Так же изменились имена каталогов, содержащих импортируемые библиотеки: каталог lib\i386\free переименован в libfre\i386, а каталог lib\i386\checked теперь называется libchk\i386.

Лично я не уверен, что в подобной перетасовке и переименовании действительно есть смысл, однако, видимо, Microsoft считает иначе и теперь все мы вынуждены мириться с этим.

Листинг 3.1. Прототип функции DriverEntry()

```
NTSTATUS DriverEntry (PDRIVER_OBJECT pDriverObject,
 PUNICODE_STRING pusRegistryPath);
```

Листинг 3.2. Определения характерных для Win32 типов данных

```
typedef UCHAR   BYTE,    *PBYTE;
typedef USHORT  WORD,    *PWORD;
typedef ULONG   DWORD,   *PDWORD;
```

Настраиваемый мастер создания драйверов

Серьезной проблемой, с которой приходится сталкиваться при разработке драйверов режима ядра, является то обстоятельство, что в состав пакета Visual C\C++ не входит мастер, позволяющий создавать проекты такого характера. Ни один из многочисленных типов проектов, предлагаемых в диалоговом окне File (файл) ▸ New (создать), не подходит для создания драйверов. К счастью, библиотека документации Microsoft Developer Network (MSDN) содержит несколько отличных статей, посвященных разработке драйверов режима ядра для Windows NT. Все они написаны Редигером Эшем (Ruediger R. Asche) в 1994 и 1995 годах. В двух из них (Asche, R. «Using the Windows NT Custom Driver Wizard». Redmond, WA: Microsoft Corporation, 1995a и Asche, R. «Wizard Simplify Windows NT Kernel-Mode Driver Design». Redmond, WA: Microsoft Corporation, 1995b) содержатся подробные инструкции о том, как добавить в Visual C/C++ мастер создания драйверов с примерами исходного кода и комментариями. Эти статьи оказали мне неоценимую помощь. Нельзя сказать, что предложенный Эшем мастер во всех отношениях удовлетворял всем моим требованиям, однако я использовал его в качестве отправной точки при создании собственного мастера. Предлагаемый мной мастер основан на разработке Редигера Эша и предназначен для формирования исходного кода драйвера ядра для Windows 2000.

Мой мастер создания драйвера вместе с полным исходным кодом содержится на прилагаемом к книге компакт-диске в каталоге \src\w2k_wiz. Ознакомившись с исходными файлами, вы обнаружите, что дословно мастер называется SBS Windows 2000 Code Wizard. На самом деле предлагаемый мастер является многоцелевым генератором заготовок программ самых разнообразных типов, включая библиотеки

Win32 DLL и прикладные программы Win32. Однако записанные на компакт-диск конфигурационные файлы настроены таким образом, что по команде пользователя мастер создает заготовку исходного кода драйвера режима ядра. Фактически предлагаемый мною мастер является преобразователем файлов, который читает с диска набор файлов, преобразует их в соответствии с набором простых правил и записывает полученный результат в виде другого набора файлов. Исходные файлы — это шаблоны, на основе которых мастер создает файлы проекта С. Изменив набор шаблонов вы можете сделать из моего мастера, например, мастера создания заготовок исходного кода библиотек DLL и т. п. Допускается использовать до семи шаблонов (если один из них отсутствует, происходит некритическая ошибка):

- Файлы с расширением .tw являются шаблонами рабочего пространства (workplace). На основе этих файлов мастер создает файлы рабочего пространства Microsoft Developer Studio Workplace Files с расширением .dsw. Скорее всего, этот тип файлов вам хорошо знаком, так как именно такие файлы необходимо указывать в диалоговом окне File (файл) ▸ Open Workplace (открыть рабочее место).
- Файлы с расширением .tp являются шаблонами проекта. На основе этих файлов мастер создает файлы проекта Microsoft Developer Studio Project Files с расширением .dsp. Файл проекта содержит конфигурационные параметры компиляции и компоновки исполняемого файла любой разновидности (Release или Debug). Ссылка на файл проекта содержится в соответствующем файле рабочего пространства.
- Файлы с расширениями .tc, .th, .tr и .td являются исходными файлами С. Мастер преобразует их в файлы с расширениями .c, .h, .rc и .def. Я надеюсь, что каждый, читающий эту книгу, знает, для чего предназначены все эти файлы.
- Файлы с расширением .ti являются файлами, содержащими графическое изображение значка. Эти файлы без изменений копируются в создаваемый проект с расширением .ico. Файл .ti содержит значок, по умолчанию включаемый в проект для того, чтобы компилятор ресурсов не выдавал ошибку. После того как мастер завершит работу, вы можете либо модифицировать этот значок, либо заменить его своим собственным.

Набор файлов, включающий в себя семь перечисленных составляющих, является необходимым минимумом для того, чтобы создать новый проект. Файл с расширением .def — это старый способ экспорта функций API из динамических библиотек DLL. Все же этот способ нравится мне больше, чем метод __declspec(dllexport). Однако в большинстве случаев драйверы не экспортируют функций, поэтому я не включил в состав мастера шаблон .td. В результате в процессе работы мастер выдает ошибку, однако это не мешает ему завершить формирование нового проекта. С тем же успехом я мог бы не включать в состав мастера шаблон описания ресурсов и файл значка, однако, как показывает практика, включение в состав нового проекта и того и другого никогда не бывает лишним. Напротив, файл .rc, по умолчанию формируемый мастером, является полноценным файлом ресурсов, помимо прочего, содержащим в себе подробную информацию о вас как о разработчике драйвера.

Правила конвертации, применяемые к набору шаблонов в процессе формирования нового проекта, просты и определяются перечнем подставляемых в шаблоны строк. Сканируя файл шаблона, мастер ищет специальные служебные последовательности символов, каждая из которых состоит из двух символов, первый из которых является символом процента. Если, просматривая шаблон, мастер встречает символ процента, он анализирует следующий за ним символ и определяет, какое действие необходимо выполнить в отношении шаблона. Служебные последовательности, распознаваемые мастером, перечисляются в табл. 3.1.

Таблица 3.1. Правила подстановки строк в файлы шаблонов

Входная последовательность	Выходная строка
%n	Имя проекта (нормальное написание)
%N	Имя проекта (все буквы прописные)
%s	Полностью определенный (fully qualified) путь к файлу w2k_wiz.ini
%d	Сегодняшний день (всегда два числа)
%m	Текущий месяц (всегда два числа)
%y	Текущий год (всегда четыре числа)
%t	Описание проекта по умолчанию, указанное в файле w2k_wiz.ini
%c	Имя компании, в которой работает автор проекта, указанное в файле w2k_wiz.ini
%a	Имя автора проекта, указанное в файле w2k_wiz.ini
%e	Адрес электронной почты автора проекта, указанный в файле w2k_wiz.ini
%p	Префикс ProgID по умолчанию, указанный в файле w2k_wiz.ini
%i	Путь к каталогу заголовочных файлов DDK, указанный в файле w2k_wiz.ini
%l	Путь к каталогу библиотечных файлов DDK, указанный в файле w2k_wiz.ini (release — окончательная версия)
%L	Путь к каталогу библиотечных файлов DDK, указанный в файле w2k_wiz.ini (debug — отладочная версия)
%%	% (используется для того, чтобы вставить в текст результирующего файла обычный знак процента)
%<другое>	Без изменений копируется в результирующий файл

В табл. 3.1 можно обнаружить несколько ссылок на файл w2k_wiz.ini. Сведения, содержащиеся в этом файле по умолчанию, показаны в примере 3.1. Как нетрудно догадаться, в строке Text файла w2k_wiz.ini необходимо указать наименование создаваемого проекта, в строке Company — имя компании, в которой работает автор проекта, в строке Email — адрес электронной почты автора, в строке Prefix — префикс ProgID, используемый автором. В строках Include, Free и Checked указываются имена служебных каталогов DDK (каталог заголовочных файлов и два каталога библиотек), а в строке Root — ключ реестра, в котором хранится имя каталога проектов Visual C/C++. Прежде чем приступить к использованию мастера, скопируйте файлы w2k_wiz.exe, w2k_wiz.ini, а также шаблоны w2k_wiz.t* из каталога \src\w2k_wiz\release на жесткий диск вашего компьютера и отредактируйте конфигурационный файл

w2k_wiz.ini таким образом, чтобы он содержал ваши индивидуальные данные. Для этого замените значения между символами < и > данными, описывающими создаваемый вами проект. Значения Include, Free и Checked необходимо настроить в соответствии с конфигурацией установленного на вашем компьютере пакета DDK. Если вы используете Visual C/C++ Version 6.0, строку Root можно оставить без изменений. Если вы используете другую версию Visual C/C++, в данной строке укажите ключ реестра, в котором хранится имя каталога, содержащего файлы проектов. Если в конце строки Root стоит символ обратной косой (\), мастер будет искать имя каталога проектов в значении по умолчанию ключа, имя которого указано в данной строке конфигурационного файла. В противном случае последовательность символов, расположенная за самым последним из символов обратной косой, расценивается мастером как имя одного из значений ключа. В примере 3.1 имя каталога проектов Visual C/C++ хранится в значении WorkplaceDir ключа HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Directories.

Пример 3.1. Индивидуальные конфигурационные параметры проекта, создаваемого при помощи мастера w2k_wiz

```
; w2k_wiz.ini
; 08-27-2000 Sven B. Schreiber
; sbs@orgon.com

[Settings]
Text       = <SBS Windows 2000 Code Wizard Project>
Company    = <MyCompany>
Author     = <MyName>
Email      = <my@mail>
Prefix     = <MyPrefix>
Include    = E:\NTDDK\inc
Free       = E:\NTDDK\libfre\i386
Checked    = E:\NTDDK\libchk\i386
Root       =  HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Directories\WorkplaceDir
```

Чтобы воспользоваться мастером, достаточно в командной строке набрать команду w2k_wiz MyDriver и нажать на Enter. По этой команде в текущем каталоге мастер создаст папку проекта с именем MyDriver, в которой будут расположены файлы MyDriver.dsw, MyDriver.dsp, MyDriver.c, MyDriver.h, MyDriver.rc и MyDriver.ico. При запуске мастера помимо имени проекта можно указать также путь к файлам проекта, и тогда папка проекта будет создана в указанном вами месте. Также перед именем проекта можно поставить символ звездочки, например w2k_wiz *MyDriver. В этом случае мастер создаст папку проекта не в текущем каталоге, а в каталоге, имя которого хранится в ключе реестра, указанном в строке Root файла w2k_wiz.ini. Иначе говоря, мастер создаст папку проекта в каталоге, который по умолчанию используется пакетом Visual C/C++ для хранения файлов проектов. Эта возможность, на мой взгляд, является наиболее удобным режимом работы мастера. Именно ее я использую чаще всего.

Мастер всегда пытается обнаружить свой конфигурационный файл и файлы шаблонов в том же каталоге, в котором расположен его исполняемый файл. Таким образом, на своем жестком диске вы можете хранить несколько копий мастера, каждая из которых обладает собственной конфигурацией. Для этого необходимо

хранить все эти копии в разных каталогах. На прилагаемом к данной книге компакт-диске хранится мастер, настроенный на создание простого проекта драйвера режима ядра, однако вы можете перенастроить его на создание программ самых разных категорий. Например, вы можете хранить на жестком диске своего компьютера копии мастера, настроенные на создание проектов драйверов, приложений Win32, динамических библиотек DLL, а также любого другого кода Windows 2000, который вы собираетесь разрабатывать.

Запуск мастера

Давайте попробуем запустить мастер создания проекта драйвера. Откройте окно приглашения командной строки Windows 2000 и введите команду w2k_wiz *TestDrv. В результате выполнения этой команды мастер создаст проект с именем TestDrv в каталоге, который по умолчанию используется пакетом Visual C/C++ для хранения проектов. Диагностические сообщения, выдаваемые мастером в процессе преобразования шаблонов, показаны в примере 3.2.

Пример 3.2. Запуск мастера генерации проектов Windows 2000

```
D:\>w2k_wiz *TestDrv

// w2k_wiz.exe
// SBS Windows 2000 Code Wizard V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Project D:\Program Files\DevStudio\MyProjects\TestDrv\

Loading D:\etc32\w2k_wiz.tc ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.c ... OK

Loading D:\etc32\w2k_wiz.td ... ERROR

Loading D:\etc32\w2k_wiz.th ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.h ... OK

Loading D:\etc32\w2k_wiz.ti ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.ico ... OK

Loading D:\etc32\w2k_wiz.tp ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsp ... OK

Loading D:\etc32\w2k_wiz.tr ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.rc ... OK

Loading D:\etc32\w2k_wiz.tw ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsw ... OK
```

Как видно, все операции выполнены успешно, за исключением преобразования шаблона .td в файл определений .def. При выполнении этой операции возникла некритическая ошибка. Дело в том, что заготовка проекта драйвера, создаваемая при помощи мастера, не нуждается в DEF-файле, поэтому шаблон .td не нужен. После того как мастер завершил свою работу, вы можете открыть новое создан-

ное им рабочее пространство при помощи команды File (файл) ▸ Open (открыть) ▸ Workspace (рабочее пространство) главного меню Visual C/C++. Вы обнаружите новую папку с именем TestDrv, в которой располагается файл рабочего пространства TestDrv.dsw, который без проблем загружается в Visual C/C++. Открыв рабочее пространство, выберите активную конфигурацию исполняемого файла. DSP-файл, сформированный мастером, определяет две возможные конфигурации:

1. Win2K kernel-mode driver (debug) — драйвер режима ядра Windows 2000 (отладочная версия).
2. Win2K kernel-mode driver (release) — драйвер режима ядра Windows 2000 (окончательная версия).

По умолчанию выбрана отладочная конфигурация, однако в любое время вы можете изменить конфигурацию исполняемого файла. Для этого необходимо воспользоваться командой Build (компоновка) ▸ Set Active Configuration (настроить активную конфигурацию) главного меню Visual C/C++. Определив конфигурацию исполняемого файла, скопируйте файл \src\common\include\DrvInfo.h с компакт-диска в любой из каталогов заголовочных файлов Visual C/C++ и откройте файлы TestDrv.c, TestDrv.h и TestDrv.rc для редактирования. Открывая файл TestDrv.rc, убедитесь в том, что вы открываете его в формате текстового файла (рис. 3.3), так как этот файл использует сложные макросы из файла DrvInfo.h, в результате чего при открытии его в редакторе ресурсов система выдает сообщение об исключении. Эта неприятная проблема впервые появилась в Visual C/C++ 5.0, и Microsoft до сих пор не удосужилась ее исправить. К счастью, в отличие от редактора ресурсов компилятор ресурсов отлично справляется с обработкой сложных макросов и не выдает никаких ошибок.

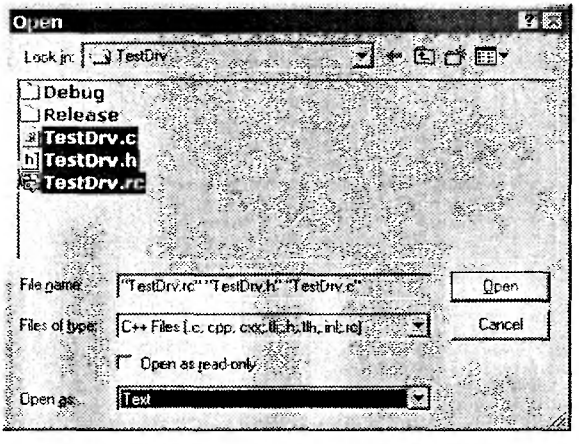

Рис. 3.3. Открытие исходных файлов драйвера в текстовом режиме

Наконец все готово к компиляции и компоновке. В главном меню Visual C/C++ выберите команду Build (компоновка) ▸ Rebuild All (скомпоновать все). Судя по сообщениям, которые отображаются на экране (пример 3.3), среда разработки без каких-либо проблем создала исполняемый файл драйвера.

В одном из подкаталогов (debug или release в зависимости от выбранной конфигурации) папки проекта компоновщик создал исполняемый файл с именем TestDrv.sys. Окончательная версия тестового драйвера занимает 5,5 Кбайт, в то время как отладочная версия занимает 8 Кбайт. Чтобы убедиться в том, что полученный в результате компиляции файл TestDrv.sys действительно содержит код и данные, вы можете воспользоваться визуальным дизассемблером MFVDasm (Multi-Format Visual Disassembler) или программой просмотра исполняемых файлов PEview (PE and COFF File Viewer). Обе эти программы записаны на прилагаемом к книге компакт-диске.

Пример 3.3. Компиляция и компоновка окончательной (release) версии тестового драйвера

```
Deleting intermediate files and output files for project 'TestDrv - Win2K ...
-------- Configuration: TestDrv - Win2K kernel-mode driver (release) ...
Compiling resources...
Compiling...
TestDrv.c
Linking...

TestDrv.sys - 0 error(s), 0 warning(s)
```

Внутри созданного мастером скелета драйвера

Содержимое файла TestDrv.c, созданного мастером, представлено в листинге 3.3. Связанный с ним заголовочный файл TestDrv.h представлен в листинге 3.4. Обратите внимание на метки <MyCompany> и <MyName> в первых строках листинга 3.3. Если в конфигурационном файле w2k_wiz.ini правильно заполнены графы имени автора и имени компании, соответствующая информация будет внесена в начало файла TestDrv.c. Также обратите внимание на то, что в начале файла указывается дата его создания, а чуть далее — история изменений и модификаций (Revision History). Рассматриваемый файл TestDrv.c был сгенерирован 27 августа 2000 года, поэтому дата указана правильно. Дополнительные сведения, извлеченные из конфигурационного файла мастера, можно обнаружить в разделе PROGRAM IDENTIFICATION листинга 3.4.

Листинг 3.3. Исходный код драйвера, сгенерированный мастером w2k_wiz

```
// TestDrv.c
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

#define  _TESTDRV_SYS_
#include <ddk\ntddk.h>
#include "TestDrv.h"

//   ===================================================================
// DISCLAIMER (ПРЕДУПРЕЖДЕНИЕ - ОТКАЗ ОТ ОТВЕТСТВЕННОСТИ)
//   ===================================================================
/*

This software is provided "as is" and any express or implied
warranties, including, but not limited to, the implied warranties of
merchantability and fitness for a particular purpose are disclaimed.
```

```
In no event shall the author <MyName> be liable for any
direct, indirect, incidental, special, exemplary, or consequential
damages (including, but not limited to, procurement of substitute
goods or services; loss of use, data, or profits; or business
interruption) however caused and on any theory of liability,
whether in contract, strict liability, or tort (including negligence
or otherwise) arising in any way out of the use of this software,
even if advised of the possibility of such damage.

*/

/*
Данное программное обеспечение поставляется в том виде,
в котором оно есть, без каких-либо гарантий, как явных,
так и подразумеваемых, включая
торговые гарантии и гарантии предназначения для той или иной
цели, но не ограничиваясь этими гарантиями.
Автор данного программного обеспечения, <MyName>,
ни при каких обстоятельствах не несет никакой ответственности
за любой ущерб, который вызван или связан с использованием
данного программного обеспечения, включая нарушение
функционирования систем, потерю данных, потерю прибыли или
нарушение работы предприятия, но не ограничиваясь этим.
*/

//   ===================================================================
// REVISION HISTORY (ИСТОРИЯ МОДИФИКАЦИЙ)
//   ===================================================================

/*

08-27-2000 V1.00 Original version.

*/

//   ===================================================================
// GLOBAL DATA (ГЛОБАЛЬНЫЕ ДАННЫЕ)
//   ===================================================================

PRESET_UNICODE_STRING (usDeviceName, CSTRING (DRV_DEVICE));
PRESET_UNICODE_STRING (usSymbolicLinkName, CSTRING (DRV_LINK  ));

PDEVICE_OBJECT  gpDeviceObject  = NULL;
PDEVICE_CONTEXT gpDeviceContext = NULL;

//   ===================================================================
// DISCARDABLE FUNCTIONS
// (ФУНКЦИИ, УДАЛЯЕМЫЕ ИЗ ПАМЯТИ ПОСЛЕ ЗАПУСКА ДРАЙВЕРА)
//   ===================================================================

NTSTATUS DriverInitialize (PDRIVER_OBJECT     pDriverObject,
                           PUNICODE_STRING    pusRegistryPath);

NTSTATUS DriverEntry (PDRIVER_OBJECT     pDriverObject,
                      PUNICODE_STRING    pusRegistryPath);

//   -------------------------------------------------------------------
```

продолжение ⇩

Листинг 3.3 (*продолжение*)

```
#ifdef ALLOC_PRAGMA

#pragma alloc_text (INIT, DriverInitialize)
#pragma alloc_text (INIT, DriverEntry)

#endif

//  ============================================================
// DEVICE REQUEST HANDLER (ОБРАБОТЧИК ЗАПРОСОВ К УСТРОЙСТВУ)
//  ============================================================

NTSTATUS DeviceDispatcher(PDEVICE_CONTEXT    pDeviceContext,
                                        PIRP                    pIrp)
 {
 PIO_STACK_LOCATION pisl;
 DWORD                      dInfo    = 0;
 NTSTATUS                ns   = STATUS_NOT_IMPLEMENTED;

 pisl = IoGetCurrentIrpStackLocation (pIrp);

 switch (pisl->MajorFunction)
     {
     case IRP_MJ_CREATE:
     case IRP_MJ_CLEANUP:
     case IRP_MJ_CLOSE:
          {
          ns = STATUS_SUCCESS;
          break;
          }
     }
 pIrp->IoStatus.Status        = ns;
 pIrp->IoStatus.Information   = dInfo;

 IoCompleteRequest (pIrp, IO_NO_INCREMENT);
 return ns;
 }

//  ============================================================
// DRIVER REQUEST HANDLER (ОБРАБОТЧИК ЗАПРОСОВ К ДРАЙВЕРУ)
//  ============================================================

NTSTATUS DriverDispatcher(PDEVICE_OBJECT    pDeviceObject,
                                        PIRP                    pIrp)
 {
 return (pDeviceObject == gpDeviceObject
          ? DeviceDispatcher (gpDeviceContext, pIrp)
          : STATUS_INVALID_PARAMETER_1);
 }

//  ------------------------------------------------------------

void DriverUnload (PDRIVER_OBJECT pDriverObject)
 {
 IoDeleteSymbolicLink     (&usSymbolicLinkName);
 IoDeleteDevice           (gpDeviceObject);
 return;
 }
```

```
//  ============================================================
// DRIVER INITIALIZATION (ИНИЦИАЛИЗАЦИЯ ДРАЙВЕРА)
//  ============================================================

NTSTATUS  DriverInitialize (PDRIVER_OBJECT      pDriverObject,
                            PUNICODE_STRING     pusRegistryPath)
  {
  PDEVICE_OBJECT pDeviceObject = NULL;
  NTSTATUS           ns = STATUS_DEVICE_CONFIGURATION_ERROR;

  if ((ns = IoCreateDevice (pDriverObject, DEVICE_CONTEXT_,
                    &usDeviceName, FILE_DEVICE_CUSTOM,
                    0, FALSE, &pDeviceObject))
       == STATUS_SUCCESS)
       {
       if ((ns = IoCreateSymbolicLink (&usSymbolicLinkName,
                                       &usDeviceName))
            == STATUS_SUCCESS)
            {
            gpDeviceObject  = pDeviceObject;
            gpDeviceContext = pDeviceObject->DeviceExtension;

            gpDeviceContext->pDriverObject = pDriverObject;
            gpDeviceContext->pDeviceObject = pDeviceObject;
            }
       else
            {
            IoDeleteDevice (pDeviceObject);
            }
       }
  return ns;
  }

//  ------------------------------------------------------------

NTSTATUS DriverEntry (PDRIVER_OBJECT      pDriverObject,
                      PUNICODE_STRING     pusRegistryPath)
  {
  PDRIVER_DISPATCH     *ppdd;
  NTSTATUS             ns = STATUS_DEVICE_CONFIGURATION_ERROR;

  if ((ns = DriverInitialize (pDriverObject, pusRegistryPath))
       == STATUS_SUCCESS)
       {
       ppdd = pDriverObject->MajorFunction;

       ppdd [IRP_MJ_CREATE                    ] =
       ppdd [IRP_MJ_CREATE_NAMED_PIPE         ] =
       ppdd [IRP_MJ_CLOSE                     ] =
       ppdd [IRP_MJ_READ                      ] =
       ppdd [IRP_MJ_WRITE                     ] =
       ppdd [IRP_MJ_QUERY_INFORMATION         ] =
       ppdd [IRP_MJ_SET_INFORMATION           ] =
       ppdd [IRP_MJ_QUERY_EA                  ] =
       ppdd [IRP_MJ_SET_EA                    ] =
       ppdd [IRP_MJ_FLUSH_BUFFERS             ] =
```

продолжение

Листинг 3.3 (*продолжение*)

```
        ppdd  [IRP_MJ_QUERY_VOLUME_INFORMATION      ] =
        ppdd  [IRP_MJ_SET_VOLUME_INFORMATION        ] =
        ppdd  [IRP_MJ_DIRECTORY_CONTROL             ] =
        ppdd  [IRP_MJ_FILE_SYSTEM_CONTROL           ] =
        ppdd  [IRP_MJ_DEVICE_CONTROL                ] =
        ppdd  [IRP_MJ_INTERNAL_DEVICE_CONTROL       ] =
        ppdd  [IRP_MJ_SHUTDOWN                      ] =
        ppdd  [IRP_MJ_LOCK_CONTROL                  ] =
        ppdd  [IRP_MJ_CLEANUP                       ] =
        ppdd  [IRP_MJ_CREATE_MAILSLOT               ] =
        ppdd  [IRP_MJ_QUERY_SECURITY                ] =
        ppdd  [IRP_MJ_SET_SECURITY                  ] =
        ppdd  [IRP_MJ_POWER                         ] =
        ppdd  [IRP_MJ_SYSTEM_CONTROL                ] =
        ppdd  [IRP_MJ_DEVICE_CHANGE                 ] =
        ppdd  [IRP_MJ_QUERY_QUOTA                   ] =
        ppdd  [IRP_MJ_SET_QUOTA                     ] =
        ppdd  [IRP_MJ_PNP                           ] = DriverDispatcher;
        pDriverObject->DriverUnload                   = DriverUnload;
        }
 return ns;
 }

//   ===================================================================
// END OF PROGRAM (КОНЕЦ ПРОГРАММЫ)
//   ===================================================================
```

Листинг 3.4. Заголовочный файл драйвера, сгенерированный мастером w2k_wiz

```
// TestDrv.h
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

//   ===================================================================
// PROGRAM IDENTIFICATION (ИДЕНТИФИКАЦИОННЫЕ ДАННЫЕ ПРОГРАММЫ)
//   ===================================================================

#define DRV_BUILD        1
#define DRV_VERSION_HIGH 1
#define DRV_VERSION_LOW  0

//   -------------------------------------------------------------------

#define DRV_DAY          27
#define DRV_MONTH        08
#define DRV_YEAR         2000

//   -------------------------------------------------------------------
// Параметры данного раздела настраиваются при помощи
//    конфигурационного файла D:\etc32\w2k_wiz.ini

#define DRV_MODULE       TestDrv
#define DRV_NAME         <SBS Windows 2000 Code Wizard Project>
#define DRV_COMPANY      <MyCompany>
#define DRV_AUTHOR       <MyName>
#define DRV_EMAIL        <my@mail>
#define DRV_PREFIX       <MyPrefix>
```

```
//  ============================================================
// HEADER FILES (ЗАГОЛОВОЧНЫЕ ФАЙЛЫ)
//  ============================================================

#include <drvinfo.h>      // здесь определяются дополнительные константы вида DRV_*

////////////////////////////////////////////////////////////////
#ifndef _RC_PASS_
////////////////////////////////////////////////////////////////

//  ============================================================
// CONSTANTS (КОНСТАНТЫ)
//  ============================================================

#define FILE_DEVICE_CUSTOM       0x8000

//  ============================================================
// STRUCTURES (СТРУКТУРЫ)
//  ============================================================

typedef struct _DEVICE_CONTEXT
 {
 PDRIVER_OBJECT  pDriverObject;
 PDEVICE_OBJECT  pDeviceObject;
 }
 DEVICE_CONTEXT, *PDEVICE_CONTEXT, **PPDEVICE_CONTEXT;

#define DEVICE_CONTEXT_ sizeof (DEVICE_CONTEXT)

////////////////////////////////////////////////////////////////
#endif // #ifndef _RC_PASS_
////////////////////////////////////////////////////////////////

//  ============================================================
// END OF FILE (КОНЕЦ ФАЙЛА)
//  ============================================================
```

Исходный код, написанный на языке C и представленный в листингах 3.3 и 3.4, — это код, являющийся общим для всех драйверов режима ядра, написанных мною до сего момента. Я спроектировал работу мастера таким образом, чтобы сделать его как можно более конфигурируемым и перенастраиваемым. При необходимости вы можете смело модифицировать предложенные мной шаблоны так, чтобы получаемый в результате работы мастера скелет драйвера обладал бы какими-либо дополнительными возможностями. Для тех, кто пока не готов вносить в предлагаемый мною образец драйвера какие-либо серьезные изменения, я предлагаю краткое описание внутренностей предлагаемой мною заготовки драйвера.

Основной точкой входа в драйвер является функция DriverEntry(). Как и в случае с точками входа в другие модули Windows 2000, наименование функции может быть другим. Вы можете присвоить этой функции любое удобное для вас символьное имя, однако при этом вы должны сообщить имя точки входа компоновщику. Для этого следует использовать ключ /switch командной строки компоновщика. В рассматриваемом нами примере соответствующий параметр компоновщика уже настроен должным образом. Чтобы убедиться в этом, изучите содержимое шаблона

w2k_wiz.tp или полученного в результате работы мастера результирующего файла TestDrv.dsp. Вы без труда обнаружите запись /entry:"DriverEntry@8" в двух местах любого из этих файлов — в соответствии с числом конфигураций исполняемого файла (свободная и проверенная). Суффикс @8 указывает на то, что функция DriverEntry() принимает в качестве аргументов восемь байт, передаваемых ей через стек. Это отлично сочетается с прототипом этой функции, определенным в листинге 3.1: в качестве аргументов функция DriverEntry() принимает два указателя, каждый из которых является 32-битным значением. В сумме получается 64 бита, что составляет 8 байт.

Прежде всего функция DriverEntry() обращается к функции DriverInitialize(), которая создает объект устройства и символическую ссылку. Символическая ссылка потребуется позже для осуществления обмена данными между устройством и приложениями пользовательского режима. Имена, используемые при обращении к вызовам IoCreateDevice() и IoCreateSymbolicLink(), формируются при помощи макросов, определенных в общем заголовочном файле DrvInfo.h, который можно обнаружить в каталоге \src\common\include на прилагаемом к книге компакт-диске. Этот файл является заголовочным файлом, формирующим самые разнообразные сведения о программе на основе нескольких базовых, настраиваемых вручную строк. Чтобы более подробно ознакомиться с этим приемом, изучите содержимое раздела PROGRAM IDENTIFICATION в файле TestDrv.h (этот раздел располагается в начале листинга 3.4). Обратите внимание на определения DRV_* и проследите, каким образом они группируются внутри файла DrvInfo.h. В частности, ресурс VERSIONINFO формируется из нескольких кусков. Среди прочих констант определяются константы DRV_DEVICE и DRV_LINK, которые в рассматриваемом примере равны соответственно \Device\TestDrv и \DosDevices\TestDrv. Обратите внимание на то, что многие функции API ядра, такие как, например, IoCreateDevice() и IoCreateSymbolicLink(), работают со строками символов, оформленными в виде специальных структур UNICODE_STRING. В данной книге структура UNICODE_STRING впервые упоминается в главе 2, на всякий случай я повторю ее определение в листинге 3.5. Макрос PRESET_UNICODE_STRING, определенный в заголовочном файле DrvInfo.h и используемый в разделе GLOBAL DATA файла TestDrv.c в листинге 3.3, создает статическую структуру UNICODE_STRING на основе простой строки Unicode. Это общепринятый упрощенный способ записи определения структур UNICODE_STRING, остающихся неизменными на протяжении времени функционирования программы.

Листинг 3.5. Стандартная структура Windows 2000: UNICODE_STRING

```
typedef struct _UNICODE_STRING
 {
 WORD Length;
 WORD MaximumLendth;
 PWORD Buffer;
 }
 UNICODE_STRING, *PUNICODE_STRING;
```

После успешного создания объекта устройства и соответствующей ему символической ссылки функция DriverInitialize() сохраняет указатели на объект устройства и контекст устройства в статических глобальных переменных. Контекст устрой-

ства — это индивидуальная структура устройства, которая может иметь произвольный размер и произвольную форму. В рассматриваемом примере тестовому устройству ставится в соответствие простая структура DEVICE_CONTEXT, определенная в файле TestDrv.h. Эта структура не содержит ничего, кроме указателей на объекты устройства и драйвера. Если разрабатываемый вами драйвер нуждается в месте постоянного хранения каких-либо необходимых для его функционирования данных, вы можете расширить эту структуру. Контекст устройства передается системой в составе каждого пакета IRP (I/O Request Packet), принимаемого драйвером.

После того как функция DriverInitialize() успешно выполняет все возложенные на нее функции и возвращает управление в функцию DriverEntry(), происходит настройка важного массива, передаваемого системой в составе структуры объекта драйвера pDriverObject. Массив предназначен для хранения указателей на функции обработки пакетов IRP различных типов. Каждый элемент массива соответствует некоторому типу пакетов IRP. Если драйвер выполняет обработку IPR-пакетов того или иного типа, функция DriverEntry() обязана внести в соответствующую этому типу позицию данного массива указатель на функцию драйвера, предназначенную для обработки пакетов IRP этого типа. Таким образом, в составе массива содержатся указатели на функции обработки всех типов запросов, на которые должен отвечать драйвер. Всего в массиве указателей на функции-обработчики содержится 28 элементов. Описание элементов этого массива содержится в табл. 3.2. Рассматриваемый скелет драйвера заносит во все 28 элементов массива указатель на одну и ту же функцию DriverDispatcher(). Ожидается, что функция DriverDispatcher() определяет, какие типы IRP представляют интерес для драйвера. Кроме того, для всех неподдерживаемых IRP эта функция возвращает значение STATUS_NOT_IMPLEMENTED. Обратите внимание, что строение массива указателей на функции-обработчики IRP в Windows NT 4.0 и Windows 2000 несколько отличается. В табл. 3.2 отличающиеся элементы массива выделены жирным шрифтом.

Таблица 3.2. Строение массива указателей на обработчики IRP в NT 4.0 и Windows 2000

Номер	Windows NT 4.0	Windows 2000
0x00	IRP_MJ_CREATE	IRP_MJ_CREATE
0x01	IRP_MJ_CREATE_NAMED_PIPE	IRP_MJ_CREATE_NAMED_PIPE
0x02	IRP_MJ_CLOSE	IRP_MJ_CLOSE
0x03	IRP_MJ_READ	IRP_MJ_READ
0x04	IRP_MJ_WRITE	IRP_MJ_WRITE
0x05	IRP_MJ_QUERY_INFORMATION	IRP_MJ_QUERY_INFORMATION
0x06	IRP_MJ_SET_INFORMATION	IRP_MJ_SET_INFORMATION
0x07	IRP_MJ_QUERY_EA	IRP_MJ_QUERY_EA
0x08	IRP_MJ_SET_EA	IRP_MJ_SET_EA
0x09	IRP_MJ_FLUSH_BUFFERS	IRP_MJ_FLUSH_BUFFERS
0x0A	IRP_MJ_QUERY_VOLUME_INFORMATION	IRP_MJ_QUERY_VOLUME_INFORMATION
0x0B	IRP_MJ_SET_VOLUME_INFORMATION	IRP_MJ_SET_VOLUME_INFORMATION
0x0C	IRP_MJ_DIRECTORY_CONTROL	IRP_MJ_DIRECTORY_CONTROL

продолжение

Таблица 3.2 (*продолжение*)

Номер	Windows NT 4.0	Windows 2000
0x0D	IRP_MJ_FILE_SYSTEM_CONTROL	IRP_MJ_FILE_SYSTEM_CONTROL
0x0E	IRP_MJ_DEVICE_CONTROL	IRP_MJ_DEVICE_CONTROL
0x0F	IRP_MJ_INTERNAL_DEVICE_CONTROL	IRP_MJ_INTERNAL_DEVICE_CONTROL
0x10	IRP_MJ_SHUTDOWN	IRP_MJ_SHUTDOWN
0x11	IRP_MJ_LOCK_CONTROL	IRP_MJ_LOCK_CONTROL
0x12	IRP_MJ_CLEANUP	IRP_MJ_CLEANUP
0x13	IRP_MJ_CREATE_MAILSLOT	IRP_MJ_CREATE_MAILSLOT
0x14	IRP_MJ_QUERY_SECURITY	IRP_MJ_QUERY_SECURITY
0x15	IRP_MJ_SET_SECURITY	IRP_MJ_SET_SECURITY
0x16	**IRP_MJ_QUERY_POWER**	**IRP_MJ_POWER**
0x17	**IRP_MJ_SET_POWER**	**IRP_MJ_SYSTEM_CONTROL**
0x18	IRP_MJ_DEVICE_CHANGE	IRP_MJ_DEVICE_CHANGE
0x19	IRP_MJ_QUERY_QUOTA	IRP_MJ_QUERY_QUOTA
0x1A	IRP_MJ_SET_QUOTA	IRP_MJ_SET_QUOTA
0x1B	**IRP_MJ_PNP_POWER**	**IRP_MJ_PNP**

После заполнения массива указателей на обработчики IRP функция DriverEntry() записывает указатель на функцию DriverUnload() драйвера в структуру объекта драйвера. Благодаря наличию этой функции система может выгрузить драйвер прямо в процессе работы ОС. Функция DriverUnload() просто уничтожает все объекты, созданные функцией DriverInitialize(). Точнее говоря, уничтожаются объект символической ссылки и объект устройства. После этого драйвер может быть безопасно удален из системы.

Функция DriverDispatcher() вызывается каждый раз, когда какой-либо модуль обращается к драйверу с запросом. Один драйвер может обслуживать несколько устройств, поэтому функция DriverDispatcher() прежде всего проверяет, какое из устройств должно обслужить запрос. Рассматриваемый скелет драйвера поддерживает работу только с одним устройством, поэтому в рамках функции осуществляется простая проверка корректности, в ходе которой указатель на объект устройства сравнивается с аналогичным указателем, полученным от функции IoCreateDevice() в процессе инициализации. Если они совпадают, функция DriverDispatcher() перенаправляет принятый пакет IRP функции DeviceDispatcher(). Помимо пакета IRP в функцию DeviceDispatcher() передается также контекст устройства, подготовленный функцией DriverInitialize(). Функция DeviceDispatcher() является функцией распределения пакетов IRP для конкретного устройства. Если вы планируете модернизировать предлагаемый скелет драйвера таким образом, чтобы он поддерживал работу с несколькими устройствами, возможно для каждого из устройств вы захотите написать отдельную, отличающуюся от остальных функцию распределения пакетов IRP. Функция DeviceDispatcher(), текст которой приведен в листинге 3.3, демонстрирует самый тривиальный вариант реализации. Она распознает всего три стандартных и самых распространенных запроса: IRP_MJ_CREATE, IRP_MJ_CLEANUP и IRP_MJ_CLOSE. В результате обработки этих запросов функция просто возвращает значение STATUS_SUCCESS. Поддержка этих запросов необходима для того, чтобы

устройство можно было открыть и закрыть без ошибок. В ответ на другие запросы драйвер возвращает значение STATUS_NOT_IMPLEMENTED.

Обратите внимание на директивы #pragma alloc_text в разделе DISCARDABLE FUNCTIONS листинга 3.3. Директива #pragma — это чрезвычайно мощное средство управления работой компилятора и компоновщика в процессе формирования исполняемого кода. Команда alloc_text предписывает разместить код указанной в директиве функции в нестандартном разделе исполняемого файла. По умолчанию весь исполняемый код заносится в раздел .text. Однако директива #pragma alloc_text (INIT, DriverEntry) предписывает компилятору и компоновщику внести код функции DriverEntry() в новый раздел исполняемого файла под названием INIT. Модуль загрузки драйверов распознает этот раздел и после завершения загрузки и начальной инициализации драйвера освобождает занимаемую этим разделом оперативную память. Дело в том, что функции DriverEntry() и DriverInitialize() вызываются только один раз — в начале работы драйвера, поэтому после того, как выполнение этих функций завершено, их можно безболезненно удалить из памяти.

Еще одной составной частью рассматриваемого скелета драйвера является сценарий ресурсов TestDrv.rc, текст которого показан в листинге 3.6. Понять его строение совсем несложно, так как этот файл состоит из ссылок на макросы, определенные в файле DrvInfo.h. Макрос DRV_RC_VERSION формирует ресурс VERSIONINFO, в состав которого входят разнообразные сведения, предоставленные мастеру, а макрос DRV_RC_ICON формирует простое выражение ICON, которое добавляет значок TestDrv.ico в раздел ресурсов исполняемого файла TestDrv.sys.

Листинг 3.6. Сценарий ресурсов скелета драйвера

```
// TestDrv.rc
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

#define _RC_PASS
#define _TESTDRV_SYS_
#include "TestDrv.h"

//  ============================================================
// STANDARD RESOURCES (СТАНДАРТНЫЕ РЕСУРСЫ)
//  ============================================================

DRV_RC_VERSION
DRV_RC_ICON

//  ============================================================
// END OF FILE (КОНЕЦ ФАЙЛА)
//  ============================================================
```

Управление вводом/выводом устройств

Как было сказано в начале главы, в данной книге мы не будем рассматривать вопросы, связанные с разработкой аппаратных драйверов. Вместо этого мы будем использовать широкие возможности драйверов режима ядра для того, чтобы изучать внутренние секреты Windows 2000. Широта возможностей драйверов режи-

ма ядра определяется тем фактом, что такие драйверы являются программными модулями, работающими на самом высоком уровне привилегий центрального процессора. Это означает, что драйвер режима ядра обладает возможностью доступа ко всем системным ресурсам, может обращаться к любым ячейкам памяти и выполнять привилегированные процессорные инструкции, например читать текущие значения служебных регистров ЦПУ. Если приложение пользовательского режима попытается прочитать хотя бы один байт из памяти, принадлежащей ядру, или выполнить инструкцию наподобие `MOV EAX, CR3`, работа этого приложения будет немедленно прервана. Драйвер режима ядра обладает существенно более широкими полномочиями, однако оборотной стороной этого является отсутствие механизмов надежной защиты, предотвращающих подвисание всей системы в случае некорректных действий со стороны драйвера. Другими словами, драйвер режима ядра, обладающий широкими полномочиями в отношении системных ресурсов, может с легкостью нарушить работоспособность всей системы. Даже самая небольшая ошибка, допущенная при разработке драйвера, может привести к появлению голубого экрана смерти (BSOD). Таким образом, разработчик драйверов режима ядра должен быть особенно внимательным, так как допущенные им ошибки обойдутся пользователям системы дороже, чем ошибки, допущенные разработчиком приложений Win32 или динамических библиотек DLL. Помните устройство-убийцу Windows 2000, которое использовалось нами в главе 1 для того, чтобы получить снимок оперативной памяти системы в момент сбоя? Чтобы произошел серьезный сбой ОС, достаточно было обратиться по виртуальному адресу 0x00000000 — одно обращение, и система не может продолжать работу! Если вы приступаете к разработке драйверов режима ядра, будьте готовы к тому, что вам придется достаточно часто перезагружать ваш компьютер.

Для того чтобы позволить приложениям пользовательского режима работать с системными ресурсами, доступными только в режиме ядра, можно использовать технологию под названием IOCTL (Device I/O Control — управление вводом/выводом устройства). Именно эта методика используется в драйверном коде, который будет рассматриваться в последующих главах. Если приложение нуждается в доступе к системному ресурсу, который недоступен в пользовательском режиме, оно обращается к драйверу режима ядра, который выполняет необходимые действия от лица приложения. Для взаимодействия между приложением и драйвером используется IOCTL. На самом деле IOCTL применяется не только в Windows 2000. Многие другие новые, равно как и давно устаревшие операционные системы также обладают механизмами IOCTL. Например, в такой древней операционной системе, как DOS 2.11, в качестве основного механизма IOCTL использовалась функция 0x44 и ее многочисленные подфункции. В своей основе IOCTL — это средство взаимодействия с устройством через канал управления, который логически отделен от канала данных. Представьте себе жесткий диск, который занят передачей содержимого дискового сектора через основной канал данных. Если клиент хочет получить информацию о текущем носителе данных, он должен воспользоваться другим каналом. Например, при помощи под-подфункции 0x66 подфункции 0x0D функции 0x44 операционной системы DOS можно прочитать 32-битный серийный номер жесткого диска (Р. Браун (R. Brown)

и Дж. Кайл (J. Kyle). *PC Interrupts: A Programmer's Reference to BIOS, DOS and Third-Party Calls.* Reading, MA: Addison Walley, 1991, 1994).

Технология IOCTL может быть реализована разными способами в зависимости от управляемого устройства, однако вне зависимости от реализации механизм IOCTL обладает следующими свойствами:

- Клиент управляет устройством, используя специальную точку входа. В DOS для этой цели использовалась функция 0x44 прерывания INT 21h. В Windows 2000 точкой входа IOCTL является функция `DeviceIoControl()`, входящая в состав Win32 API и экспортируемая модулем `kernel32.dll`.
- Обращаясь к функции, являющейся точкой входа IOCTL, клиент передает ей идентификатор устройства, число, являющееся кодом управления, буфер вводимых данных и буфер выводимых данных. В Windows 2000 идентификатором устройства является переменная типа `HANDLE`, то есть дескриптор успешно открытого устройства.
- При помощи кода управления (control code) клиентское приложение сообщает механизму IOCTL о том, какая функция должна быть выполнена в отношении устройства от лица клиента.
- Буфер вводимых данных содержит любые дополнительные данные, которые могут потребоваться устройству для того, чтобы обработать запрос клиента.
- Если в результате обработки запроса устройство генерирует какие-либо данные, эти данные размещаются в буфере выводимых данных.
- О результате выполнения операции IOCTL клиент узнает при помощи кода состояния.

Очевидно, что IOCTL — это чрезвычайно мощный многоцелевой механизм, при помощи которого можно организовать обработку самых разнообразных запросов. Например, приложение пользовательского режима может воспользоваться технологией IOCTL для того, чтобы получить доступ к памяти, принадлежащей ядру. Если приложение попытается обратиться к такой памяти напрямую, возникнет исключение и работа приложения будет прервана. Чтобы обойти защитные механизмы ОС, приложение может загрузить в память драйвер режима ядра, который будет работать с памятью ядра от лица приложения. Управление драйвером будет осуществляться при помощи IOCTL. Чтобы осуществлять передачу данных, оба модуля должны использовать один и тот же протокол IOCTL. Это означает, что они должны договориться о значении управляющих кодов и о размещении данных в буферах ввода и вывода. Например, если приложение хочет читать содержимое памяти ядра, оно пересылает драйверу управляющий код 0x80002000, если же оно намерено осуществить запись данных в память ядра, оно использует управляющий код 0x80002001. Если запрос является запросом на чтение, в буфере ввода IOCTL, скорее всего, будут содержаться базовый адрес и размер в байтах участка памяти, который требуется прочитать. Приняв запрос, драйвер режима ядра прежде всего определяет, какую операцию (чтение или запись) необходимо выполнить в отношении памяти ядра. Для этого он анализирует код управления. Определив, что принятый запрос является запросом на чтение, драйвер копирует

интересующий клиента участок памяти в буфер вывода IOCTL. Если буфер вывода обладает размером, достаточным для того, чтобы вместить интересующие клиента данные, драйвер возвращает вызвавшему приложению код состояния, указывающий на успешное завершение операции. Аналогично этому при обработке запроса на запись данных в память ядра драйвер копирует данные из буфера ввода в область памяти, которая определяется базовым адресом и количеством байт, которые также хранятся в буфере ввода. В главе 4 будет рассмотрен исходный код подобного драйвера, осуществляющего слежение за памятью ядра.

Таким образом, технология IOCTL играет роль черного хода, при помощи которого приложения Win32 могут выполнять фактически любые действия, которые в нормальных условиях могут выполняться только модулями привилегированных режимов. Конечно же, для того чтобы реализовать эту возможность на практике, вам прежде всего потребуется написать соответствующий драйвер ядра, который сможет выполнять привилегированные операции от имени приложений пользовательского режима, однако если в системе уже работает подобный модуль, работа с ресурсами на уровне ядра не будет представлять для вас сложностей. В данной книге мы рассмотрим, каким образом осуществляется разработка подобного кода, кроме того, будет рассмотрен пример драйвера, при помощи которого вы сможете делать воистину удивительные вещи.

Устройство-убийца Windows 2000

Прежде чем переходить к более сложным проектам, рассмотрим очень простой драйвер. В главе 1 я продемонстрировал устройство-убийцу Windows 2000 — драйвер w2k_kill.sys, вызывающий сбой системы. Данный драйвер не нуждается в большей части кода листинга 3.3, так как он нарушает работу системы еще до того, как у него появляется возможность принимать пакеты IRP. Реализация драйвера продемонстрирована в листинге 3.7. Содержимое файла w2k_kill.h здесь не перепечатывается, так как в этом файле не содержится никакого интересного для нас кода.

Код листинга 3.7 не пытается выполнить какую-либо инициализацию внутри функции DriverEntry(). Система останавливается до того, как функция DriverEntry() возвращает управление, поэтому никакой дополнительной работы не требуется.

Листинг 3.7. Небольшой драйвер, вызывающий сбой системы

```
#define  _W2K_KILL_SYS_
#include <ddk\ntddk.h>
#include "w2k_kill.h"

//=============================================================
// DISCARDABLE FUNCTIONS
// (ФУНКЦИИ, УДАЛЯЕМЫЕ ИЗ ПАМЯТИ ПОСЛЕ ЗАПУСКА ДРАЙВЕРА)
//=============================================================

NTSTATUS OriverEntry (PDRIVER_OBJECT          pDriverObject,
                      PUNICODE_STRING         pusRegistryPath);

#ifdef ALLOC_PRAGMA
#pragma alloc_text (INIT, DriverEntry)
#endif
```

```
//   ================================================================
// DRIVER INITIALIZATION (ИНИЦИАЛИЗАЦИЯ ДРАЙВЕРА)
//   ================================================================

NTSTATUS  OriverEntry (PDRIVER_OBJECT       pDriverObject,
                       PUNICODE_STRING      pusRegistryPath)
 {
 return *((NTSTATUS *) 0);
 }

//   ================================================================
// END OF PROGRAM (КОНЕЦ ПРОГРАММЫ)
//   ================================================================
```

Загрузка и выгрузка драйверов

Разработав драйвер ядра, вы, скорее всего, захотите немедленно запустить его. Как это сделать? Как правило, драйверы загружаются в память и начинают работу в процессе начальной загрузки операционной системы, но неужели вам придется перезагружать систему каждый раз, когда вы желаете протестировать очередную версию вашего драйвера? К счастью, для того чтобы добавить драйвер в систему, вовсе не обязательно перезагружать ее. Windows 2000 поддерживает специальный интерфейс, позволяющий загружать и выгружать драйверы прямо в процессе работы операционной системы. Для загрузки и выгрузки драйверов используется диспетчер управления службами SC Manager (Service Control Manager). В последующих разделах работа этого механизма будет рассмотрена подробнее.

Диспетчер управления службами

Наименование Service Control Manager (диспетчер управления службами) не соответствует действительности, так как можно сделать вывод, что этот механизм служит только лишь для управления службами. Службы — это разновидность модулей Windows 2000, которые хорошо адаптированы для выполнения прикладных задач в фоновом режиме вне зависимости от оболочки графического пользовательского интерфейса. Говоря иначе, служба — это процесс Win32, который может продолжать функционирование в системе, даже если к ней не подключен ни один пользователь. Разработка служб — весьма интересная тема для обсуждения, однако в данной книге мы не будет заниматься рассмотрением связанных с этим вопросов. Более подробно о разработке служб можно узнать, ознакомившись с замечательным руководством Паулы Томлинсон (Paula Tomplinson), опубликованным в журнале «Windows Developer's Journal» (WDJ). Полное название этой публикации Tomlinson P. «How To Write NT Service». «Windows Developer's Journal 7», no. 2, Feb 1996. San Francisco, CA: CMP Media. Кроме того, множество интересных сведений о службах NT содержится в колонке «Understanding NT», которую Паула вела в WDJ в 1996 году.

На самом деле SC Manager работает как со службами, так и с драйверами. Чтобы упростить повествование, далее я буду использовать термин «служба» для обозначения любых объектов, с которыми работает SC Manager. К этим объектам от-

носятся как службы в общепринятом смысле этого слова, так и драйверы режима ядра. Интерфейс SC доступен для приложений Win32 в составе компонента advapi32.dll, который является частью подсистемы Win32. В составе этого компонента содержится интересный набор API-функций. Имена основных функций, используемых для того, чтобы загружать и выгружать службы, а также осуществлять управление службами, перечислены в табл. 3.3. Там же вы найдете краткое описание этих функций. Прежде чем вы сможете работать с интерфейсом SC, вы должны получить дескриптор диспетчера служб. Для этого необходимо обратиться к функции OpenSCManager(). В дальнейшем я буду называть этот дескриптор *дескриптором диспетчера* (manager handle). Этот дескриптор необходимо использовать при обращении к функциям CreateService() и OpenService(). В свою очередь, каждая из упомянутых функций также возвращает дескриптор, который я буду называть *дескриптором службы* (service handle). Дескрипторы такого типа необходимо использовать при обращении к вызовам, имеющим отношение к конкретной службе. К подобным вызовам относятся функции ControlService(), DeleteService() и StartService(). Для освобождения дескрипторов обоих типов используется вызов CloseServiceHandle().

Таблица 3.3. Важные функции управления службами

Имя	Описание
CloseServiceHandle	Закрывает дескриптор, полученный при помощи функций OpenSCManager(), CreateService() или OpenService()
ControlService	Используется для того, чтобы остановить, приостановить, продолжить работу, а также опросить или оповестить загруженные в память драйвер или службу
CreateService	Загрузить службу/драйвер
DeleteService	Выгрузить службу/драйвер
OpenSCManager	Получить дескриптор диспетчера SC Manager
OpenService	Получить дескриптор загруженной службы/драйвера
QueryServiceStatus	Получить сведения о свойствах и текущем состоянии службы/драйвера
StartService	Запустить загруженную службу/драйвер

Загрузка и запуск службы подразумевает выполнение следующей последовательности действий:

1. Обращение к OpenSCManager() для получения дескриптора диспетчера.
2. Обращение к CreateService() для того, чтобы добавить службу в систему.
3. Обращение к StartService() для того, чтобы перевести службу в состояние функционирования.
4. Обращение к CloseServiceHandle() для того, чтобы освободить дескрипторы диспетчера и службы.

Если в процессе выполнения этой последовательности действий на одном из шагов возникла ошибка, вы должны обязательно позаботиться о том, чтобы вернуть систему в исходное состояние. Иначе говоря, вы должны выполнить действия, обратные тем, которые вы успели выполнить до того, как возникла ошибка. Напри-

мер, если при обращении к вызову StartService() диспетчер SC Manager выдает ошибку, вы должны обратиться к DeleteService() для того, чтобы удалить службу из системы. В противном случае служба сохранится в системе и будет находиться в нежелательном состоянии. Еще одной тонкостью, которую следует учитывать при работе с интерфейсом SC Manager, является то обстоятельство, что при обращении к функции CreateService() вы должны передать ей полное имя исполняемого файла службы. Если вы укажете путь к файлу относительно текущего каталога, функция не сможет загрузить службу, так как она не будет искать файл в текущем каталоге. По этой причине если вы не используете полных имен файлов, при обращении к функции CreateService() вы должны сформировать такое имя. Для этого можно использовать функцию GetFullPathName(), которая является частью стандартного Win32 API.

Высокоуровневые функции управления драйверами

Чтобы облегчить взаимодействие с SC Manager, вы можете воспользоваться записанными на прилагаемом к данной книге компакт-диске вспомогательными функциями. Эти функции играют роль оболочки интерфейса SC Manager и скрывают от использующего их программиста большую часть странностей и неудобств, свойственных этому интерфейсу. Предлагаемые функции являются частью большой библиотеки Windows 2000 Utility Library, которая располагается на компакт-диске в каталоге \src\w2k_lib. Имена всех функций, экспортируемых библиотекой w2k_lib.dll, начинаются с префикса w2k, а имена функций, имеющих отношение к управлению драйверами и службами, начинаются с префикса w2kService. В листинге 3.8 показана реализация библиотечных функций, которые предназначены для загрузки и выгрузки драйверов и служб, а также управления ими.

Листинг 3.8. Библиотечные функции управления службами и драйверами

```
SC_HANDLE WINAPI w2kServiceConnect (void)
 {
 return OpenSCManager (NULL, NULL, SC_MANAGER_ALL_ACCESS);
 }

//  ------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceDisconnect (SC_HANDLE hManager)
 {
 if (hManager != NULL) CloseServiceHandle (hManager);
 return NULL;
 }

//  ------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceManager (SC_HANDLE     hManager,
                                    PSC_HANDLE    phManager,
                                    BOOL          fOpen)

 {
 SC_HANDLE hManager1 = NULL;
```

продолжение

Листинг 3.8 (*продолжение*)

```
 if (phManager != NULL)
      {
      if (fOpen)
           {
           if (hManager == NULL)
                {
                *phManager = w2kServiceConnect ();
                }
           else
                {
                *phManager = hManager;
                }
           }
      else
           {
           if (hManager == NULL)
                {
                *phManager = w2kServiceDisconnect (*phManager);
                }
           }
      hManager1 = *phManager;
      }
 return hManager1;
 }

//  ------------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceOpen     (SC_HANDLE      hManager,
                                     PWORD          pwName)
 {
 SC_HANDLE hManager1;
 SC_HANDLE hService = NULL;

 w2kServiceManager (hManager, &hManager1, TRUE);

 if ((hManager1 != NULL) && (pwName != NULL))
      {
      hService = OpenService (hManager1, pwName,
                              SERVICE_ALL_ACCESS);
      }
 w2kServiceManager (hManager, &hManager1, FALSE);
 return hService;
 }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceClose (SC_HANDLE hService)
 {
 return (hService != NULL) && CloseServiceHandle (hService);
 }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceAdd (SC_HANDLE      hManager,
                           PWORD          pwName,
                           PWORD                  pwInfo,
                           PWORD                  pwPath)
```

```
{
SC_HANDLE hManager1, hService;
PWORD     pwFile;
WORD  awPath [MAX_PATH];
DWORD     n;
BOOL  fOk = FALSE;

w2kServiceManager (hManager, &hManager1, TRUE);

if ((hManager1 != NULL) && (pwName != NULL) &&
    (pwInfo   != NULL) && (pwPath != NULL) &&
    (n = GetFullPathName (pwPath, MAX_PATH, awPath, &pwFile)) &&
    (n < MAX_PATH))
    {
    if ((hService = CreateService(hManager1, pwName, pwInfo,
                                  SERVICE_ALL_ACCESS,
                                  SERVICE_KERNEL_DRIVER,
                                  SERVICE_DEMAND_START,
                                  SERVICE_ERROR_NORMAL,
                                  awPath, NULL, NULL,
                                  NULL, NULL, NULL))
        != NULL)
        {
        w2kServiceClose (hService);
        fOk = TRUE;
        }
    else
        {
        fOk =    (GetLastError () ==
              ERROR_SERVICE_EXISTS);
        }
    }
w2kServiceManager (hManager, &hManager1, FALSE);
return fOk;
}

//  ---------------------------------------------------------------------

BOOL WINAPI w2kServiceRemove  (SC_HANDLE    hManager,
                               PWORD        pwName)
{
SC_HANDLE hService;
BOOL   fOk = FALSE;

if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
    {
    if (DeleteService (hService))
        {
        fOk = TRUE;
        }
    else
        {
        fOk =     (GetLastError () ==
              ERROR_SERVICE_MARKED_FOR_DELETE);
        }
    w2kServiceClose (hService);
    }
```

продолжение ⇩

Листинг 3.8 (*продолжение*)

```
 return fOk;
 }

//  ---------------------------------------------------------------

BOOL WINAPI w2kServiceStart    (SC_HANDLE     hManager,
                                PWORD         pwName)
 {
 SC_HANDLE hService;
 BOOL   fOk = FALSE;

 if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
      {
      if (StartService (hService, 1, &pwName))
           {
           fOk = TRUE;
           }
      else
           {
           fOk =     (GetLastError () ==
                ERROR_SERVICE_ALREADY_RUNNING);
           }
      w2kServiceClose (hService);
      }
 return fOk;
 }

//  ---------------------------------------------------------------

BOOL WINAPI w2kServiceControl (SC_HANDLE     hManager,
                               PWORD         pwName,
                               DWORD         dControl)
 {
 SC_HANDLE          hService;
 SERVICE_STATUS     ServiceStatus;
 BOOL               fOk = FALSE;

 if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
      {
      if (QueryServiceStatus (hService, &ServiceStatus))
           {
           switch  (ServiceStatus.dwCurrentState)
                {
                case  SERVICE_STOP_PENDING:
                case  SERVICE_STOPPED:
                     {
                     fOk = (dControl == SERVICE_CONTROL_STOP);
                     break;
                     }
                case  SERVICE_PAUSE_PENDING:
                case  SERVICE_PAUSED:
                     {
                     fOk = (dControl == SERVICE_CONTROL_PAUSE);
                     break;
                     }
                case  SERVICE_START_PENDING:
                case  SERVICE_CONTINUE_PENDING:
                case  SERVICE_RUNNING:
```

```
                    {
                    fOk = (dControl == SERVICE_CONTROL_CONTINUE);
                    break;
                    }
                }
            }
        fOk = fOk ||
              ControlService (hService, dControl, &ServiceStatus);

        w2kServiceClose (hService);
        }
    return fOk;
    }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceStop     (SC_HANDLE      hManager,
                                PWORD          pwName)
    {
    return w2kServiceControl(hManager, pwName,
                             SERVICE_CONTROL_STOP);
    }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServicePause    (SC_HANDLE      hManager,
                                PWORD          pwName)
    {
    return w2kServiceControl(hManager, pwName,
                             SERVICE_CONTROL_PAUSE);
    }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceContinue (SC_HANDLE      hManager,
                                PWORD          pwName)
    {
    return w2kServiceControl(hManager, pwName,
                             SERVICE_CONTROL_CONTINUE);
    }

//  ------------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceLoad     (PWORD  pwName,
                                     PWDRD  pwInfo,
                                     PWDRD  pwPath,
                                     BOOL   fStart)
    {
    BOOL   fOk;
    SC_HANDLE hManager = NULL;

    if ((hManager = w2kServiceConnect ()) != NULL)
        {
        fOk = w2kServiceAdd (hManager, pwName, pwInfo, pwPath);

        if (fOk && fStart)
            {
            if (!(fOk = w2kServiceStart (hManager, pwName)))
```

продолжение ⇩

Листинг 3.8 (*продолжение*)

```
                {
                w2kServiceRemove (hManager, pwName);
                }
            }
        if (!fOk)
            {
            hManager = w2kServiceDisconnect (hManager);
            }
        }
    return hManager;
    }

//  ------------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceLoadEx  (PWORD pwPath,
                                    BOOL  fStart)
    {
    PVS_VERSIONDATA     pvvd;
    PWORD               pwPath1, pwInfo;
    WORD                awName [MAX_PATH];
    DWORD               dName, dExtension;
    SC_HANDLE           hManager = NULL;

    if (pwPath != NULL)
        {
        dName = w2kPathName (pwPath, &dExtension);

        lstrcpyn  (awName, pwPath + dName,
                   min (MAX_PATH, dExtension - dName + 1));

        pwPath1 = w2kPathEvaluate (pwPath,  NULL);
        pvvd = w2kVersionData  (pwPath1, -1);

        pwInfo = ((pvvd != NULL) && pvvd->awFileDescription [0]
                        ? pvvd->awFileDescription
                        : awName);

        hManager = w2kServiceLoad (awName, pwInfo, pwPath1, fStart);

        w2kMemoryDestroy  (pvvd);
        w2kMemoryDestroy  (pwPath1);
        }
    return hManager;
    }

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceUnload  (PWORD      pwName,
                               SC_HANDLE hManager)
    {
    SC_HANDLE hManager1 = hManager;
    BOOL      fOk       = FALSE;

    if (pwName != NULL)
        {
        if (hManager1 == NULL)
```

```
        {
        hManager1 = w2kServiceConnect ();
        }
    if (hManager1 != NULL)
        {
        w2kServiceStop (hManager1, pwName);
        fOk = w2kServiceRemove (hManager1, pwName);
        }
    }
w2kServiceDisconnect (hManager1);
return fOk;
}

//  ------------------------------------------------------------------

BOOL WINAPI w2kServiceUnloadEx (PWORD          pwPath,
                                SC_HANDLE      hManager)
{
DWORD dName, dExtension;
WORD  awName [MAX_PATH];
PWORD pwName = NULL;

if (pwPath != NULL)
    {
    dName = w2kPathName (pwPath, &dExtension);

    lstrcpyn (pwName = awName, pwPath + dName,
              min (MAX_PATH, dExtension - dName + 1));
    }
return w2kServiceUnload (pwName, hManager);
}
```

Функции, определенные в листинге 3.8, перечислены в табл. 3.4 с короткими комментариями. Имена некоторых библиотечных функций напоминают имена аналогичных функций интерфейса SC Manager API. Например, аналогами библиотечных функций w2kServiceStart() и w2kServiceControl() являются функции StartService() и ControlService(), входящие в состав интерфейса SC Manager API. Это не случайно, так как функции w2kServiceStart() и w2kServiceControl() на самом деле играют роль оболочки для функций StartService() и ControlService(). Отличие состоит в том, что функции StartService() и ControlService() работают с дескрипторами служб, в то время как функции w2kServiceStart() и w2kServiceControl() идентифицируют службы при помощи имен. Имена служб незаметно для программиста преобразуются в дескрипторы соответствующих служб при помощи обращений к вызовам w2kServiceOpen() и w2kServiceClose(), которые, в свою очередь, обращаются к функциям OpenService() и CloseServiceHandle().

Таблица 3.4. Функции библиотеки w2k_lib.dll, облегчающие работу с интерфейсом SC Manager

Имя	Описание
w2kServiceAdd	Добавляет службу/драйвер в систему
w2kServiceClose	Закрывает дескриптор службы
w2kServiceConnect	Подключается к диспетчеру управления службами (Service Control Manager)

продолжение

Таблица 3.4 (*продолжение*)

Имя	Описание
w2kServiceContinue	Продолжает работу службы/драйвера, функционирование которой было приостановлено
w2kServiceControl	Останавливает, приостановливает, продолжает работу, опрашивает или оповещает загруженную в память службу/драйвер
w2kServiceDisconnect	Отключается от диспетчера управления службами (Service Control Manager)
w2kServiceLoad	Загружает и при желании запускает службу/драйвер
w2kServiceLoadEx	Загружает и при желании запускает службу/драйвер (автоматическая генерация имени)
w2kServiceManager	Открывает/закрывает временный дескриптор диспетчера SC Manager
w2kServiceOpen	Получает дескриптор загруженной службы/драйвера
w2kServicePause	Приостанавливает работу функционирующей службы/драйвера
w2kServiceRemove	Удаляет службу/драйвер из системы
w2kServiceStart	Запускает загруженную службу/драйвер
w2kServiceStop	Останавливает функционирование работающей службы/драйвера
w2kServiceUnload	Останавливает службу/драйвер и выгружает ее из памяти
w2kServiceUnloadEx	Останавливает службу/драйвер и выгружает ее из памяти (автоматическая генерация имени)

Функции из табл. 3.4 используются в соответствии со следующими правилами:

- Чтобы загрузить службу, используются вызовы w2kServiceLoad() или w2kServiceLoadEx(). Расширенная функция w2kServiceLoadEx() на основании пути к исполняемому файлу и ресурса версии автоматически генерирует имя службы и отображаемое имя. Аргумент fStart определяет, надо ли автоматически запустить службу после завершения успешной загрузки. В случае успеха каждая из функций возвращает дескриптор диспетчера для осуществления дальнейших запросов. В случае если служба уже загружена, а также в случае, если значение fStart равно TRUE, а служба уже работает, обе функции не возвращают никаких сообщений об ошибках. В случае возникновения ошибки при необходимости служба автоматически выгружается из памяти.
- Чтобы выгрузить службу из памяти, используются вызовы w2kServiceUnload() или w2kServiceUnloadEx(). При этом в качестве аргумента эти функции принимают дескриптор диспетчера, возвращаемый функцией w2kServiceLoad() или w2kServiceLoadEx(). Функция w2kServiceUnloadEx() генерирует имя службы автоматически на основании пути к исполняемому файлу. Если вы успели закрыть дескриптор диспетчера, вы можете получить новый дескриптор при помощи вызова w2kServiceConnect(). Кроме того, в качестве дескриптора диспетчера вы можете передать функции выгрузки драйвера значение NULL, и тогда будет использован временный дескриптор. Функция w2kServiceUnload() автоматически закрывает дескриптор диспетчера. Если в момент обращения к одной из этих функций служба уже помечена для удаления, но не может быть удалена по той причине, что существуют открытые дескрипторы устройств, ни одна из этих функций не возвращает никаких сообщений об ошибках.

- Для управления службой используются вызовы w2kServiceStart(), w2kServiceStop(), w2kServicePause() и w2kServiceContinue(). В качестве аргумента все эти функции принимают дескриптор диспетчера, полученный в результате обращения к вызовам w2kServiceLoad() или w2kServiceConnect(). Если в качестве дескриптора диспетчера вы передадите любой из этих функций значение NULL, будет использован временный дескриптор. Если служба уже находится в запрашиваемом состоянии, никакого сообщения об ошибке не выдается.
- Чтобы закрыть дескриптор диспетчера, обратитесь к функции w2kServiceDisconnect(). Позже, в любое удобное время вы можете получить новый дескриптор диспетчера при помощи вызова w2kServiceConnect().

Вызов w2kServiceLoadEx() является чрезвычайно мощной функцией. Чтобы загрузить службу, достаточно передать этой функции только один аргумент: путь к исполняемому файлу. Все остальные параметры генерируются автоматически. Имя службы, которое требуется для обращения к вызову CreateService() интерфейса SC Manager, получается из имени исполняемого файла службы путем удаления из него файлового расширения. Чтобы сформировать приемлемое отображаемое имя службы, функция w2kServiceLoadEx() пытается прочитать значение строки FileDescription, входящей в состав раздела сведений о версии файла. Если в составе исполняемого файла отсутствуют сведения о версии файла, значит, строка FileDescription недоступна для чтения. В этом случае в качестве отображаемого имени по умолчанию используется имя службы. В отличие от w2kServiceLoad() функция w2kServiceLoadEx() корректно обрабатывает имена переменных окружения, входящих в состав полного имени исполняемого файла загружаемой службы. Иными словами, если в составе пути к исполняемому файлу службы содержатся имена переменных %SystemRoot% или %TEMP%, вместо этих имен подставляются текущие значения соответствующих переменных окружения. Функция w2kServiceUnloadEx() является антиподом функции w2kServiceLoadEx() — из предоставленного ей пути к исполняемому файлу она извлекает имя службы и передает это имя функции w2kServiceUnload(). Обе функции (w2kServiceLoadEx() и w2kServiceUnloadEx()) удобно использовать при разработке приложений, которые от имени пользователя осуществляют загрузку и выгрузку драйверов сторонних производителей, не обладая при этом никакой информацией о драйвере, за исключением пути к его исполняемому файлу. Пример такого приложения содержится на прилагаемом к книге компакт-диске. Консольная утилита w2k_load.exe является универсальным средством загрузки и выгрузки драйверов режима ядра, которое обеспечивает простой пользовательский интерфейс командной строки для обращения к функциям w2kServiceLoadEx() и w2kServiceUnloadEx(). Исходные файлы этой утилиты содержатся в каталоге \src\w2k_load компакт-диска. Соответствующий код показан в листинге 3.9. Можно заметить, что внутреннее строение программы примитивно, так как вся наиболее тяжелая работа выполняется функциями w2kServiceLoadEx() и w2kServiceUnloadEx(), являющимися частью файла w2k_lib.dll.

Листинг 3.9. Исходный код программы загрузки/выгрузки драйверов устройств

```
#include "w2k_load.h"
//  ===========================================================
// GLOBAL STRINGS (ГЛОБАЛЬНЫЕ СТРОКИ)
//  ===========================================================

WORD awUsage  [] =
 L"\r\n"
 L"Usage: " SW(MAIN_MODULE) L" <driver path>\r\n"
 L"       " SW(MAIN_MODULE) L" <driver path> %s\r\n"
 L"       " SW(MAIN_MODULE) L" <driver name> %s\r\n";

WORD awUnload   [] = L"/unload";

WORD awOk       [] = L"OK\r\n";
WORD awError    [] = L"ERROR\r\n";

//  ===========================================================
// COMMAND HANDLERS (ОБРАБОТЧИКИ КОМАНД)
//  ===========================================================

BOOL WINAPI DriverLoad (PWORD pwPath)
 {
 SC_HANDLE hManager;
 BOOL      fOk = FALSE;

 _printf (L"\r\nLoading \"%s\" ... ", pwPath);

 if ((hManager = w2kServiceLoadEx (pwPath, TRUE)) != NULL)
     {
     w2kServiceDisconnect (hManager);
     fOk = TRUE;
     }
 _printf (fOk ? awOk : awError);
 return fOk;
 }

//  -----------------------------------------------------------

BOOL WINAPI DriverUnload (PWORD pwPath)
 {
 BOOL fOk = FALSE;

 _printf (L"\r\nUnloading \"%s\" ... ", pwPath);

 fOk = w2kServiceUnloadEx (pwPath, NULL);

 _printf (fOk ? awOk : awError);
 return fOk;
 }

//  ===========================================================
// MAIN PROGRAM (ТЕЛО ОСНОВНОЙ ПРОГРАММЫ)
//  ===========================================================

DWORD Main (DWORD argc, PTBYTE *argv, PTBYTE *argp)
 {
 _printf (atAbout);

 if (argc == 2)
```

```
        {
        DriverLoad (argv [1]);
        }
    else
        {
        if ((argc == 3) && (!lstrcmpi (argv [2]. awUnload)))
            {
            DriverUnload (argv [1]);
            }
        else
            {
            _printf (awUsage, awUnload, awUnload);
            }
        }
    return 0;
    }

//  ========================================================
// END OF PROGRAM (КОНЕЦ ПРОГРАММЫ)
//  ========================================================
```

Остальные функции, упомянутые в табл. 3.4, работают на более низком уровне и служат в основном для использования другими процедурами модуля w2k_lib.dll. Конечно же, при желании вы можете обратиться к любой из них из своего приложения. Назначение и методы использования этих функций можно понять, изучив листинг 3.8.

Последовательный опрос служб и драйверов

Время от времени возникает надобность узнать, какие службы и драйверы в настоящее время загружены в системе и в каком они находятся состоянии. Для этой цели можно использовать еще одну чрезвычайно мощную функцию интерфейса SC Manager под названием EnumServicesStatus(). Как и другие вызовы данного API, в качестве аргумента эта функция принимает дескриптор диспетчера. В процессе своей работы функция EnumServicesStatus() заполняет массив структур ENUM_SERVICE_STATUS. В результате после возвращения из этой функции данный массив содержит сведения обо всех загруженных в системе службах и драйверах. Перечень служб и драйверов, возвращаемый функцией EnumServiceStatus(), можно фильтровать в соответствии с типом и состоянием служб и драйверов. Если при обращении к этой функции вы передаете ей указатель на буфер, размер которого слишком мал для того, чтобы вместить в себя записи обо всех загруженных в системе службах и драйверах, для получения полной информации вы можете обратиться к этой функции несколько раз подряд. Каждый раз в буфере будет размещаться информация о новых загруженных в системе драйверах и службах. Сложно заранее определить размер буфера, достаточный для того, чтобы вместить информацию обо всех загруженных драйверах и службах за один раз, так как в буфере должно быть предусмотрено дополнительное место неизвестного размера для хранения строковых полей-членов структуры ENUM_SERVICE_STATUS. К счастью, функция EnumServiceStatus() возвращает количество байт, необходимое для того, чтобы разместить все структуры ENUM_SERVICE_STATUS, которые не уместились в изначально выделенном для

этой цели буфере. Таким образом, вы можете определить нужный вам размер буфера, используя метод проб и ошибок. В листинге 3.10 содержатся определения структур SERVICE_STATUS и ENUM_SERVICE_STATUS в том виде, в котором эти структуры определены в заголовочном файле WinSvc.h интерфейса Win32 API.

Листинг 3.10. Определение структур ENUM_SERVICE_STATUS и SERVICE_STATUS

```
typedef struct _SERVICE_STATUS
 {
 DWORD dwServiceType;
 DWORD dwCurrentState;
 DWORD dwControlsAccepted;
 DWORD dwWin32ExitCode;
 DWORD dwServiceSpecificExitCode;
 DWORD dwCheckPoint;
 DWORD dwWaitHint;
 }
 SERVICE_STATUS, *LPSERVICE_STATUS;

typedef struct _ENUM-SERVICE_STATUS
 {
 LPTSTR        lpServiceName;
 LPTSTR        lpDisplayName;
 SERVICE_STATUS ServiceStatus;
 }
 ENUM_SERVICE_STATUS;
```

Функция w2kServiceList(), текст которой приведен в листинге 3.11, — это еще один удобный инструмент из библиотеки w2k_lib.dll, записанной на прилагаемом к книге компакт-диске. Эта функция берет на себя выполнение всех рутинных процедур, связанных с получением перечня загруженных в системе драйверов и служб, и в качестве результата своей работы возвращает указатель на готовую к использованию структуру W2K_SERVICES, содержащую сведения обо всех загруженных в системе драйверах и службах, а кроме того, некоторую полезную дополнительную информацию. Структура W2K_SERVICES определяется в файле w2k_lib.h. Текст этого определения содержится в начале листинга 3.11. Помимо массива aess[] значений типа ENUM_SERVICE_STATUS в состав этой структуры входят четыре дополнительных члена. Поле dEntries — это количество записей, скопированных в массив aess[], а поле dBytes — это размер результирующей структуры W2K_SERVICES в байтах. Наконец, в полях dDisplayName и dServiceName содержится максимальная длина строк lpDisplayName и lpServiceName в составе записей массива aess[]. Знание максимальной длины отображаемых имен и имен служб для загруженных в системе драйверов и служб может оказаться чрезвычайно полезным, например в случае, если вы разрабатываете консольное приложение, выводящее на экран список имен загруженных драйверов/служб с выравниванием колонки имен по правому или левому краю.

Чтобы получить исчерпывающую информацию о загруженных в системе драйверах и службах и оформить все эти сведения в виде единой структуры, функция w2kServiceList() пытается получить записи обо всех загруженных драйверах и службах при помощи всего двух обращений к вызову EnumServiceStatus(). Для достижения этой цели вначале функции EnumServiceStatus() в качестве аргумента передается указатель на буфер нулевой длины. В результате возникает ошибка

ERROR_MORE_DATA, однако при этом функция EnumServiceStatus() возвращает количество байт, которое потребуется для того, чтобы разместить записи обо всех загруженных в системе службах и драйверах. Выделив буфер требуемого размера, функция w2kServiceList() обращается к вызову EnumServiceStatus() повторно. На этот раз в большинстве случаев выполнение этого вызова должно окончиться успехом. Однако существует небольшая вероятность, что между двумя обращениями к вызову EnumServiceStatus() в системе появились новая служба или драйвер. Таким образом, к списку загруженных драйверов/служб добавилась еще одна запись. Чтобы корректно обработать подобную ситуацию, описанная процедура получения списка служб осуществляется в цикле до тех пор, пока либо при выполнении вызова EnumServiceStatus() не возникнет ни одной ошибки, либо возникнет ошибка, отличающаяся от ERROR_MORE_DATA.

Листинг 3.11. Получение информации о загруженных службах и драйверах

```
typedef struct _W2K_SERVICES
 {
 DWORD          dEntries;          // количество элементов в массиве aess[]
 DWORD          dBytes;            // размер структуры в байтах
 DWORD          dDisplayName;      // максимальная длина отображаемого имени
 DWORD          dServiceName;      // максимальная длина имени службы
 ENUM_SERVICE_STATUS aess []:      // массив записей о состоянии служб/драйверов
 }
 W2K_SERVICES, *PW2K_SERVICES, **PPW2K_SERVICES;

#define W2K_SERVICES_ sizeof (W2K_SERVICES)


PW2K_SERVICES WINAPI w2kServiceList(BOOL fDriver,
                                    BOOL fWin32,
                                    BOOL fActive,
                                    BOOL fInactive)
 {
 SC_HANDLE hManager;
 DWORD          dType, dState, dBytes, dResume, dName, i;
 PW2K_SERVICES pws = NULL;

 if ((pws = w2kMemoryCreate (W2K_SERVICES_)) != NULL)
     {
     pws->dEntries           = 0;
     pws->dBytes             = 0;
     pws->dDisplayName  = 0;
     pws->dServiceName  = 0;

     if ((fDriver || fWin32) && (fActive || fInactive))
         {
         if ((hManager = w2kServiceConnect ()) != NULL)
             {
             dType      = (fDriver ? SERVICE_DRIVER : 0) |
                  (fWin32  ? SERVICE_WIN32  : 0);

             dState  = (fActive && fInactive
                           ? SERVICE_STATE_ALL
                           : (fActive
                               ? SERVICE_ACTIVE
                               : SERVICE_INACTIVE));
```

продолжение

Листинг 3.11 (*продолжение*)

```
                dBytes  = pws->dBytes;

                while (pws != NULL)
                     {
                     pws->dEntries       = 0;
                     pws->dBytes         = dBytes;
                     pws->dDisplayName   = 0;
                     pws->dServiceName   = 0;

                     dResume = 0;

                     if (EnumServicesStatus (hManager, dType, dState,
                                             pws->aess, pws->dBytes,
                                             &dBytes, &pws->dEntries,
                                             &dResume))
                          break;

                     dBytes += pws->dBytes;
                     pws    = w2kMemoryDestroy (pws);

                     if (GetLastError () != ERROR_MORE_DATA) break;

                     pws = w2kMemoryCreate (W2K_SERVICES_ + dBytes);
                     }
                w2kServiceDisconnect  (hManager);
                }
           else
                {
                pws = w2kMemoryDestroy (pws);
                }
           }
      if (pws != NULL)
           {
           for (i = 0; i < pws->dEntries; i++)
                {
                dName = lstrlen (pws->aess [i].lpDisplayName);
                pws->dDisplayName = max (pws->dDisplayName, dName);

                dName = lstrlen (pws->aess [i].lpServiceName);
                pws->dServiceName = max (pws->dServiceName, dName);
                }
           }
      }
 return pws;
 }
```

Функция w2kServiceList() принимает четыре булевых аргумента, определяющих содержимое возвращаемого списка. Аргументы fDriver и fWin32 определяют, надо ли включать в состав списка информацию о драйверах и службах (соответственно). Если значение обоих аргументов равно TRUE, значит, в результирующем списке будут содержаться как записи, относящиеся к службам, так и записи, относящиеся к драйверам. При помощи флагов fActive и fInactive вы можете фильтровать содержимое списка в соответствии с состоянием загруженных служб/драйверов. В частности, если флаг fActive установлен, значит, в результирующий список

будет включена информация обо всех модулях, которые в настоящий момент работают (running) или функционирование которых приостановлено (paused). Флаг fInactive включает в список модули, загруженные в память, но остановленные (stopped). Если все четыре аргумента содержат значение FALSE, функция возвращает структуру W2K_SERVICES с пустым массивом. На прилагаемом к данной книге компакт-диске записана простая утилита просмотра списка загруженных служб и драйверов. Эта утилита является консольным приложением Win32 и использует функцию w2kServiceList() из библиотеки w2k_lib.dll. Для того чтобы выровнять имена служб и драйверов, утилита w2k_svc.exe использует значения полей dDisplayName и dServiceName структуры W2K_SERVICES (см. листинг 3.11). Исходный код данной утилиты содержится в каталоге \src\w2k_svc компакт-диска. Запустить утилиту можно прямо с компакт-диска, используя для этой цели исполняемый файл \bin\w2k_svc.exe. Список, полученный в результате запуска этой утилиты на моем компьютере, показан в примере 3.4. В данном случае я приказал утилите вывести на экран список всех работающих драйверов режима ядра. Для этого была использована команда w2k_svc /drivers /active.

Пример 3.4. Запуск утилиты w2k_svc.exe

```
D:\> w2k_svc /drivers /active

// w2k_svc.exe
// SBS Windows 2000 Service List V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Found 29 active drivers:

  1. Alerter . . . . . . . . . . . . . . . . . . . . . . . . . . Alerter
  2. Computer Browser . . . . . . . . . . . . . . . . . . . . . Browser
  3. Creative Service for CDROM Access . . . . . . . . . . Creative Service
  4. DHCP Client . . . . . . . . . . . . . . . . . . . . . . . . Dhcp
  5. Logical Disk Manager . . . . . . . . . . . . . . . . . . . dmserver
  6. DNS Client . . . . . . . . . . . . . . . . . . . . . . . . Dnscache
  7. Event Log . . . . . . . . . . . . . . . . . . . . . . . . . Eventlog
  8. COM+ Event System . . . . . . . . . . . . . . . . . . . . EventSystem
  9. Server . . . . . . . . . . . . . . . . . . . . . . . . . . lanmanserver
 10. Workstation . . . . . . . . . . . . . . . . . . . . . . . lanmanworkstation
 11. TCP/IP NetBIOS Helper Service . . . . . . . . . . . . . . LmHosts
 12. Messenger . . . . . . . . . . . . . . . . . . . . . . . . Messenger
 13. Network Connections . . . . . . . . . . . . . . . . . . . Netman
 14. Removable Storage . . . . . . . . . . . . . . . . . . . . NtmsSvc
 15. Plug and Play . . . . . . . . . . . . . . . . . . . . . . PlugPlay
 16. IPSEC Policy Agent . . . . . . . . . . . . . . . . . . . . PolicyAgent
 17. Protected Storage . . . . . . . . . . . . . . . . . . . . ProtectedStorage
 18. Remote Access Connection Manager . . . . . . . . . . . . . RasMan
 19. Remote Registry Service . . . . . . . . . . . . . . . . . RemoteRegistry
 20. Remote Procedure Call (RPC). . . . . . . . . . . . . . . . RpcSs
 21. Security Accounts Manager . . . . . . . . . . . . . . . . SamSs
 22. Task Scheduler . . . . . . . . . . . . . . . . . . . . . . Schedule
 23. RunAs Service . . . . . . . . . . . . . . . . . . . . . . seclogon
 24. System Event Notification . . . . . . . . . . . . . . . . SENS
```

продолжение

Пример 3.4 (*продолжение*)

```
25. Print Spooler . . . . . Spooler
26. Telephony . . . . . TapiSrv
27. Distributed Link Tracking Client . . . . . TrkWks
28. Windows Management Instrumentation . . . . . WinMgmt
29. Windows Management Instrumentation Driver Extensions . Wmi
```

В следующей главе мы приступим к разработке драйвера режима ядра, который позволит просматривать память ядра и изучать содержимое важнейших структур управления памятью. Работа над проектом будет начата в главе 4 и продолжена в главах 5 и 6, и в каждой последующей главе в разрабатываемый нами драйвер будут добавляться все новые возможности. Он будет становиться все более совершенным. В результате мы получим мощное средство наблюдения за ядром Windows 2000, к которому будет приложен набор из нескольких весьма удобных клиентских приложений.

4 Исследование памяти Windows 2000

Управление памятью — одна из самых важных и самых трудных задач операционной системы. В этой главе представлены исчерпывающий обзор управления памятью в Windows 2000 и схема распределения 4 Гбайт линейного адресного пространства. Объясняются такие возможности семейства процессоров Intel i386, как адресация виртуальной памяти и управление подкачкой страниц, а также принципы работы с ними ядра Windows 2000. Для исследования памяти в этой главе рассмотрены две вспомогательные программы-примера: драйвер режима ядра, собирающий информацию о системе, и клиентское приложение пользовательского режима, которое обращается к драйверу через механизм управления вводом-выводом и выводит эти данные в окно консоли. Модуль драйвера слежения (spy driver) будет еще несколько раз использоваться в оставшихся главах для других интересных задач, в которых требуется выполнение кода в режиме ядра. Чтение этой главы, особенно первой ее части, потребует определенных усилий, поскольку в ней идет речь непосредственно об аппаратном устройстве процессора. Тем не менее я надеюсь, что вы не станете пропускать эту главу, поскольку управление виртуальной памятью — захватывающая тема, а понимание принципов организации памяти такой сложной операционной системы, как Windows 2000, поможет глубоко изучить особенности внутреннего устройства системы.

Управление памятью в процессорах Intel i386

Ядро Windows 2000 интенсивно использует механизмы управления виртуальной памятью защищенного режима процессоров класса Intel i386. Для лучшего понимания методов управления основной памятью в Windows 2000 важно хотя бы немного знать об особенностях архитектуры процессора i386. Термин *i386* может показаться устаревшим, поскольку процессор 80386 относится к раннему периоду вычислений в среде Windows, а Windows 2000 предназначена для работы на процессорах класса Pentium и выше. Однако даже в этих новейших моделях

процессоров применяется схема управления памятью, первоначально разработанная для процессора 80386, естественно, с некоторыми важными усовершенствованиями. Поэтому Microsoft обычно обозначает версии Windows NT и 2000 как созданные для процессоров Intel «i386» или даже «x86». Пусть это вас не смущает — где бы в книге вы ни встретили числа 86 или 386, помните о том, что приведенная информация относится к конкретной *архитектуре*, а не конкретной модели процессора.

Базовая модель памяти

Для приложений и системного кода Windows 2000 применяет очень простую модель памяти. Виртуальное адресное пространство размером 4 Гбайт, предлагаемое 32-разрядными процессорами Intel, делится на две равные части. Адреса памяти ниже *0x80000000* предназначены для модулей пользовательского режима, в том числе и для подсистемы Win32, а другие 2Гбайт зарезервированы для ядра. Windows 2000 Advanced Server дополнительно поддерживает еще одну модель памяти, обычно называемую *4GT RAM Tuning*, впервые представленную в Windows NT Server Enterprise Edition. В рамках этой модели пользовательским процессам отводится 3Гбайт адресного пространства, а ядру — 1 Гбайт. Эта схема активизируется при добавлении ключа /3GB в командную строку начальной загрузки в файле конфигурации системного загрузчика boot.ini.

Версии Windows 2000 Advanced Server и Datacenter Server поддерживают еще один вариант использования памяти, называемый *Physical Address Extension* (PAE), активизируемый ключом /PAE в файле boot.ini. В этом случае используется особенность некоторых процессоров Intel (например, Pentium Pro), позволяющая отображать пространство физической памяти выше 4 Гбайт в 32-разрядное адресное пространство. В этой главе я не буду касаться этих особых случаев. Дополнительный материал для чтения можно найти в статье Q171793 Microsoft Knowledge Base, руководстве Intel по процессору Pentium и документации Windows 2000 Device Driver Kit, DDK.

Сегментация памяти и подкачка страниц по запросу

Прежде чем углубляться в технические подробности архитектуры i386, давайте вернемся назад в 1978 год, когда Intel выпустила прародителя всех процессоров для персональных компьютеров: процессор 8086. Здесь я хотел бы ограничиться обсуждением только ключевых моментов. Если вас интересует дополнительная информация, справочник программиста по процессору 80486, написанный Робертом Л. Хаммелом (Robert L. Hummel), будет превосходной начальной книгой. Сегодня он уже несколько устарел, поскольку не охватывает новые возможности семейства Pentium, но зато в нем больше внимания уделяется основам архитектуры i386. Хотя процессор 8086 мог адресовать 1 Мбайт оперативной памяти (RAM, Random Access Memory), приложение никогда не «видело» все физическое адресное пространство, так как адресные регистры процессора были

ограничены 16 битами. Это означало, что приложения могли обращаться к непрерывному линейному адресному пространству объемом только в 64 Кбайт, но это окно можно было передвигать вдоль физического пространства при помощи набора 16-битных сегментных регистров. Каждый сегментный регистр задавал 16-разрядный базовый адрес, измеряемый в 16-байтных величинах (параграфах), а линейные адреса логического пространства размером 64 Кбайт добавлялись как смещение к этой базе (значение которой умножалось на 16), что давало в итоге 20-разрядные адреса физической памяти. Эта устаревшая модель адресации до сих пор поддерживается даже в самых последних процессорах Pentium, она называется режимом реальной адресации (Real Address Mode), или просто *реальным режимом* (Real Mode).

В процессоре 20286 появился альтернативный режим адресации, называемый режимом защищенной виртуальной адресации (Protected Virtual Address Mode), или *защищенным режимом* (Protected Mode). В новой модели памяти физические адреса генерировались не просто добавлением линейного адреса к сегментной базе. Для сохранения обратной совместимости с процессорами 8086 и 80186 процессор 80286 все еще использовал сегментные регистры, но после переключения процессора в защищенный режим в них хранились уже не адреса физических сегментов памяти, а селектор, в котором содержался индекс в таблице дескрипторов. Соответствующая запись в таблице определяла 24-разрядный физический базовый адрес, что давало возможность адресовать 16 Мбайт оперативной памяти, в то время казавшихся невообразимой величиной. Несмотря на это, процессор 286 оставался 16-разрядным и ограничение линейного адресного пространства в 64 Кб все еще действовало.

Прорыв был осуществлен в 1985 году с появлением процессора 80386. Наконец-то узы 16-разрядной адресации были разорваны и линейное адресное пространство расширилось до 4 Гбайт, что обеспечивалось появлением 32-разрядных линейных адресов одновременно с сохранением базовой архитектуры селектор/дескриптор предыдущего процессора. К счастью, в структуре дескриптора процессора 80286 сохранялось несколько свободных бит, которые можно было задействовать. При переходе от 16-разрядных адресов к 32-разрядным был также удвоен размер регистров данных процессора и добавлены новые эффективные режимы адресации. Кардинальный переход к 32-разрядным данным и адресам должен был предоставить программистам значительные преимущества — по крайней мере теоретически. На практике прошло еще несколько лет, прежде чем платформа Windows начала полностью поддерживать 32-разрядную архитектуру. 26 июля 1993 года вышла в свет первая версия Windows NT, содержащая самый первый вариант Win32 API. В то время как при программировании в среде Windows 3.x все еще приходилось работать с фрагментами памяти в 64 Кбайт с разделением на сегменты кода и данных, Windows NT обеспечивала плоское линейное адресное пространство в 4 Гбайт, в котором и к коду, и к данным можно было обращаться посредством простых 32-битных указателей без какой-либо сегментации. На аппаратном уровне, конечно, сегментация сохранялась, как я покажу далее в этой главе, но вся ответственность за управление сегментами была окончательно передана операционной системе.

Другой совершенно новой особенностью процессора 80386 была аппаратная поддержка подкачки страниц (paging), или, более точно, модели виртуальной памяти с подкачкой страниц по требованию (demand paging). Эта техника позволяла выгружать данные из оперативной памяти на другое устройство хранения, такое как жесткий диск. При включенной подкачке страниц процессор может обращаться к большему объему памяти, чем ее физический размер, перенося содержимое наименее используемых участков памяти на альтернативное устройство хранения, освобождая место в основной памяти для новых данных. Теоретически таким образом можно получить доступ ко всем 4 Гбайт непрерывной линейной памяти при условии наличия достаточно большого объема памяти резервного носителя — даже если имеющаяся в наличии физическая оперативная память составляет лишь небольшую часть от общего количества памяти. Разумеется, доступ к памяти при реализованной подкачке страниц происходит достаточно медленно, и неплохо бы всегда иметь максимально возможный размер физической оперативной памяти. Однако подкачка страниц — прекрасный способ работы с данными большого объема, размер которых иначе вышел бы за границы доступной памяти. Например, графическим приложениям и базам данных требуется большой объем рабочей памяти, и некоторые из них не смогут работать на компьютерах младших моделей, где подкачка страниц невозможна.

В схеме подкачки страниц процессора 80386 память делится на страницы размером 4 Кбайт или 4 Мбайт. Проектировщик операционной системы волен выбрать любую из двух возможностей, можно даже использовать страницы обоих размеров. Позже я покажу, что Windows 2000 применяет именно такую смешанную схему размера страниц, размещая операционную систему в страницах размером 4 Мбайт, а для остальных кода и данных используя страницы в 4 Кбайт. Управление страницами осуществляется при помощи иерархического дерева страниц, в котором для каждой страницы хранится информация о ее месте нахождения в физической памяти, а также о том, хранится ли страница в оперативной памяти на самом деле. Если страница была выгружена на жесткий диск и какой-либо модуль пытается обратиться к памяти в пределах этой страницы, процессор сгенерирует ошибку страницы (page fault), аналогичную прерыванию, пришедшему от периферийного аппаратного устройства. Затем находящийся в ядре операционной системы обработчик ошибки страницы попытается загрузить эту страницу обратно в физическую память, возможно, для освобождения места записывая на диск другой участок памяти. Как правило, для определения кандидатов на выгрузку операционная система применяет стратегию выталкивания дольше всего не использовавшейся страницы (least recently used, LRU). Теперь становится понятно, почему эту схему часто называют подкачкой страниц *по запросу*: фрагменты содержимого физической памяти перемещаются на резервное устройство хранения и обратно по запросу программ, основываясь на статистике использования памяти операционной системой и приложениями.

Образованный таблицами размещения страниц уровень косвенной адресации влечет за собой следующие интересные особенности. Во-первых, нет никакой предопределенной связи между используемыми программой адресами и адресами на физической адресной шине микросхемы процессора. Если известно, что структура

данных в приложении размещена, к примеру, по адресу 0x00140000, то о ее физическом адресе все равно ничего нельзя сказать без обращения к дереву размещения страниц. Право решать, на какой физический адрес будет отображен интересующий нас адрес, предоставлено операционной системе. Более того, текущая трансляция адресов на самом деле непредсказуема, отчасти благодаря вероятностной природе механизма подкачки. К счастью, большинству приложений нет необходимости знать конкретные физические адреса. Это может понадобиться разве что разработчикам драйверов оборудования. Вторая особенность подкачки страниц заключается в том, что адресное пространство не обязано быть непрерывным. В зависимости от содержимого таблицы страниц пространство в 4 Гбайт может содержать большие промежутки разрыва, на которые не отображается ни физическая память, ни память на резервном устройстве. Если приложение попытается прочитать или записать данные по этому адресу, оно будет немедленно остановлено системой. Позже в этой главе я подробно остановлюсь на том, как Windows 2000 распределяет доступную память по адресному пространству в 4 Гбайт.

В процессорах 80486 и Pentium применяются в точности такие же механизмы сегментации и подкачки страниц по схеме i386, что и в процессоре 80386, за исключением некоторых довольно экзотических особенностей адресации, таких как Phisical Address Extension (PAE) в процессоре Pentium Pro. Помимо более высокой тактовой частоты у новых моделей процессоров оптимизированы и другие характеристики. Например, в процессоре Pentium реализован двойной конвейер инструкций, позволяющий ему выполнять одновременно две операции, пока эти инструкции не зависят друг от друга. Например, если инструкция A изменяет значение регистра, а следующей инструкции B необходимо использовать для своих вычислений это измененное значение, то B не сможет быть выполнена до завершения инструкции A. Но если инструкция B использует другой регистр, процессор сможет выполнить инструкции A и B одновременно без отрицательных последствий. Эта оптимизация процессора Pentium вкупе с остальными усовершенствованиями открыла богатые возможности для оптимизации компиляторов. Если эта тема представляет для вас интерес, обратитесь к книге Рика Бута (Rick Booth) *Inner Loops* (Booth, 1997).

В рамках управления памятью i386 необходимо различать три разновидности адресов, в руководстве Intel по системному программированию для процессора Pentium (Intel 1999c) называемых *логическими*, *линейными* и *физическими* адресами.

1. *Логические адреса* (logical address). Это наиболее точное определение ячейки памяти, обычно записываемое в шестнадцатеричной форме вида XXXX:YYYYYYYY, где XXXX задает селектор, а YYYYYYYY — это линейное смещение в сегменте, на который указывает селектор. Вместо числового значения XXXX также возможно указать имя сегментного регистра, в котором хранится значение селектора, такого как CS (code segment, сегмент кода), DS (data segment, сегмент данных), ES (extra segment, дополнительный сегмент), FS (первый дополнительный сегмент данных), GS (второй дополнительный сегмент данных) и SS (stack segment, сегмент стека). Эта нотация унаследована от старого стиля «сегмент:смещение» спецификации «дальних указателей» (far pointer) в реальном режиме процессора 8086.

2. *Линейные адреса* (linear address). Большинство приложений и многие драйверы режима ядра игнорируют виртуальные адреса. Точнее, их интересует только часть смещения виртуальных адресов, которая носит название *линейного адреса*. Адрес этого типа подразумевает использование модели сегментации по умолчанию, которая задается текущими значениями сегментных регистров процессора. Windows 2000 использует плоскую модель сегментации, в которой регистры CS, DS, ES и SS указывают на одно и то же линейное адресное пространство, поэтому программы могут считать, что указатели на код, данные и стек можно безопасно приводить друг к другу. Например, указатель на стек можно в любое время привести к указателю на данные, не заботясь о значении соответствующих сегментных регистров.
3. *Физические адреса* (physical address). Этот тип адресов представляет интерес, только если процессор работает в режиме страничного обмена. По существу, физический адрес представляет собой напряжение на выводах адресной шины микросхемы процессора. Операционная система отображает линейные адреса на физические, устанавливая таблицы страниц (page-table). Схема таблиц страниц Windows 2000, обладающая весьма интересными свойствами для разработчиков отладочного программного обеспечения, будет обсуждена далее в этой главе.

Различия между виртуальными и линейными адресами довольно искусственны, и встречается документация, в которой оба термина употребляются как взаимозаменяемые. Я сделаю все возможное, чтобы все-таки последовательно использовать приведенную выше классификацию. Важно отметить, что Windows 2000 считает физические адреса 64-разрядными. Это может показаться странным в системах Intel i386, в которых, как правило, используется 32-разрядная адресная шина. Несмотря на это, некоторые процессоры Pentium могут адресовать более 4 Гбайт физической памяти. Например, в режиме Physical Address Extension (PAE) процессора Pentium Pro физическое адресное пространство расширяется до 36 бит, что дает возможность адресовать до 64 Гбайт оперативной памяти (Intel, 1999c). Поэтому функции API Windows 2000, задействующие физические адреса, как правило, полагаются на тип данных PHYSICAL_ADDRESS, который является просто псевдонимом структуры данных LARGE_INTEGER, как показано в листинге 4.1. Оба типа определены в заголовочном файле DDK ntdef.h. Объединение LARGE_INTEGER представляет 64-битное целое со знаком в виде структуры данных, позволяющей интерпретировать свое значение либо как две связанные 32-битные величины (LowPart и HighPart), либо как одно 64-битное число (QuadPart). Тип данных LONGLONG эквивалентен собственному типу данных Visual C/C++ __int64. Его беззнаковый аналог называется ULONGLONG или DWORDLONG и основан на собственном типе unsigned __int64.

На рис. 4.1 приведена схема модели сегментации i386, отображающая связи между логическими и линейными адресами. Для ясности я нарисовал таблицу дескрипторов и сегмент в виде небольших неперекрывающихся прямоугольников, что, в принципе, необязательно. В действительности 32-разрядная операционная система, как правило, применяет схему сегментации, показанную на рис. 4.2. Эта так называемая плоская модель (flat model) памяти основана на сегментах, полностью занимающих 4 Гбайт адресного пространства. В качестве побочного эффекта таблица

дескрипторов становится частью сегмента, и к ней может обратиться любой код, обладающий соответствующими правами доступа.

Листинг 4.1. Определение типов данных PHYSICAL_ADDRESS и LARGE_INTEGER

```
typedef LARGE_INTEGER PHYSICAL_ADDRESS, *PPHYSICAL_ADDRESS;

typedef union _LARGE_INTEGER
      {
      struct
           {
           ULONG     LowPart;
           LONG HighPart;
           };
      LONGLONG QuadPart;
      }
      LARGE_INTEGER, *PLARGE_INTEGER;
```

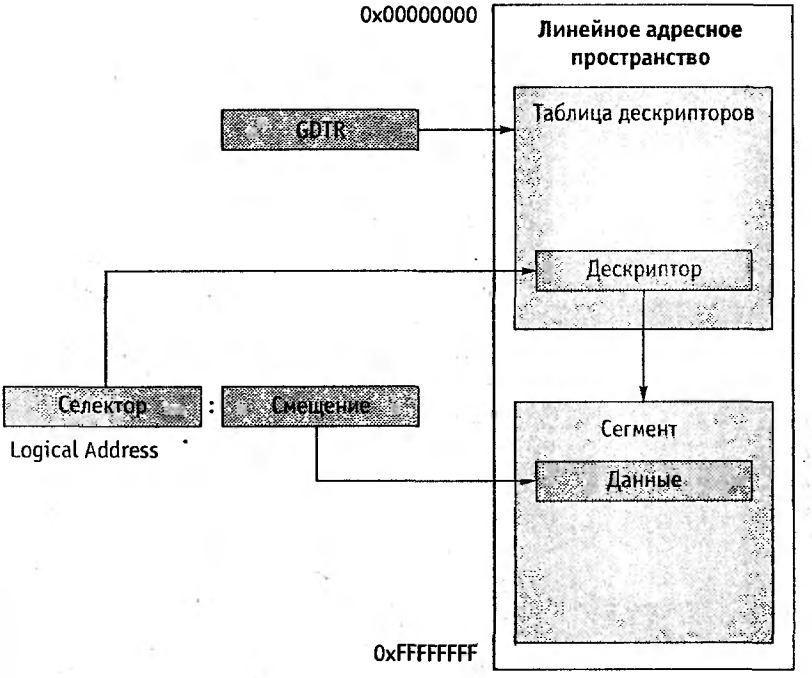

Рис. 4.1. Сегментация памяти i386

Модель памяти на рис. 4.2 применяется Windows 2000 для стандартных сегментов кода, данных и стека, то есть всех логических адресов, использующих сегментные регистры CS, DS, ES и SS. Сегменты FS и GS трактуются по-другому. Регистр GS Windows 2000 не использует, а регистр FS адресует специальные системные области данных внутри линейного адресного пространства. Поэтому его базовый адрес больше нуля, а размер меньше 4 Гбайт. Интересно, что Windows 2000 поддерживает различные сегменты FS в режиме пользователя и режиме ядра. Далее в этой главе приведен дополнительный материал по этой теме.

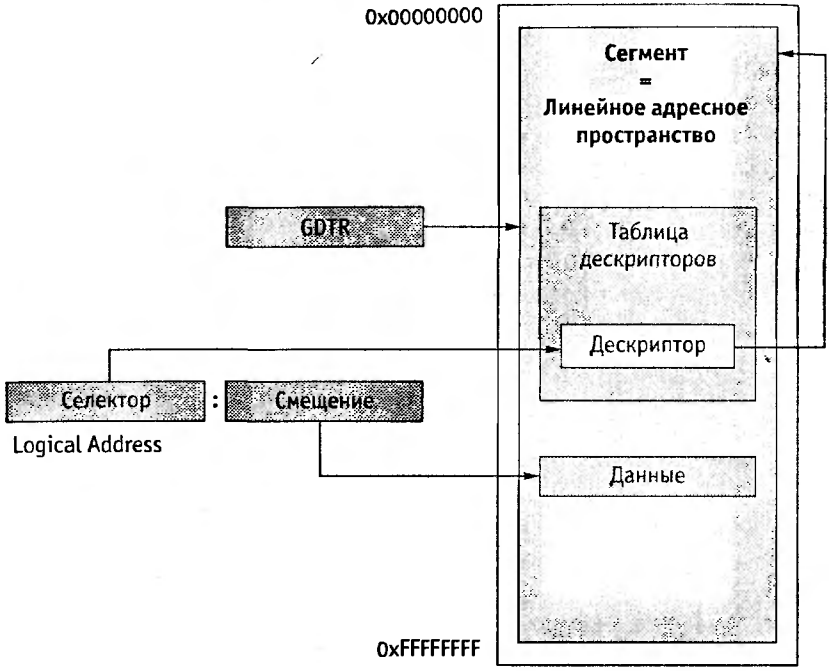

Рис. 4.2. Плоская модель сегментации 4 Гбайт памяти

На рис. 4.1 и 4.2 часть селектора логического адреса указывает на таблицу дескрипторов, заданную регистром GDTR. Это регистр глобальной таблицы дескрипторов (Global Descriptor Table Register, GDTR) процессора, который может быть установлен операционной системой в любой подходящий линейный адрес. Первая запись в глобальной таблице дескрипторов (Global Descriptor Table, GDT) зарезервирована, а соответствующий селектор называется «селектором нулевого сегмента» и предназначен для использования в качестве начального значения неиспользуемых сегментных регистров. Таблица GDT в Windows 2000 хранится по адресу 0x80036000. В таблице GDT может храниться до 8192 64-битных записей, соответственно ее максимальный размер — 64 Кбайт. Windows 2000 использует только первые 128 записей, что ограничивает размер GDT до 1024 байт. Помимо GDT в процессорах i386 существуют локальная таблица дескрипторов (Local Descriptor Table, LDT) и таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT), адресуемых соответственно регистрами LDTR и IDTR. Значения регистров GDTR и IDTR уникальны и действуют для всех выполняемых процессором задач, а значение LDTR связано с конкретной задачей и при использовании содержит 16-битный селектор GDT.

Рисунок 4.3 демонстрирует сложный механизм преобразования линейного адреса в физический, действующий в модуле управления памяти i386, если подкачка страниц по запросу включена в режиме страницы размером 4 Кбайт. Базовый регистр каталога страниц (Page Directory Base Register, PDBR) в левом верхнем углу рисунка содержит физический базовый адрес каталога страниц (page directory).

PDBR идентичен регистру i386 CR3. Для адресации используются только старшие 20 бит, поэтому каталог страниц всегда размещен на границе страницы. Остальные биты PDBR либо представляют собой флаги, либо зарезервированы для будущих расширений. Каталог страниц занимает в точности одну страницу размером 4 Кбайт. С точки зрения структуры данных каталог — это массив из 1024 32-битных записей каталога страниц (page directory entry, PDE). Аналогично регистру PBDR, каждая запись PDE делится на 20-битный номер страничного блока (page frame number, PFN), адресующего таблицу страницы, и массив битовых флагов. Каждая таблица страницы выровнена по границе страницы и занимает 4 Кбайт, в которых содержатся 1024 записи таблицы страниц (page table entry, PTE). Опять же старшие 20 бит извлекаются из записи в таблице страниц (PTE), образуя указатель на страницу данных объемом 4 Кбайт. Преобразование адресов выполняется путем разбиения линейного адреса на три части: старшие 10 бит выбирают запись PTE в таблице страниц, адресованной записью в каталоге страниц (PDE), и затем младшие 12 бит определяют смещение в странице данных, определяемой записью в таблице страниц (PTE).

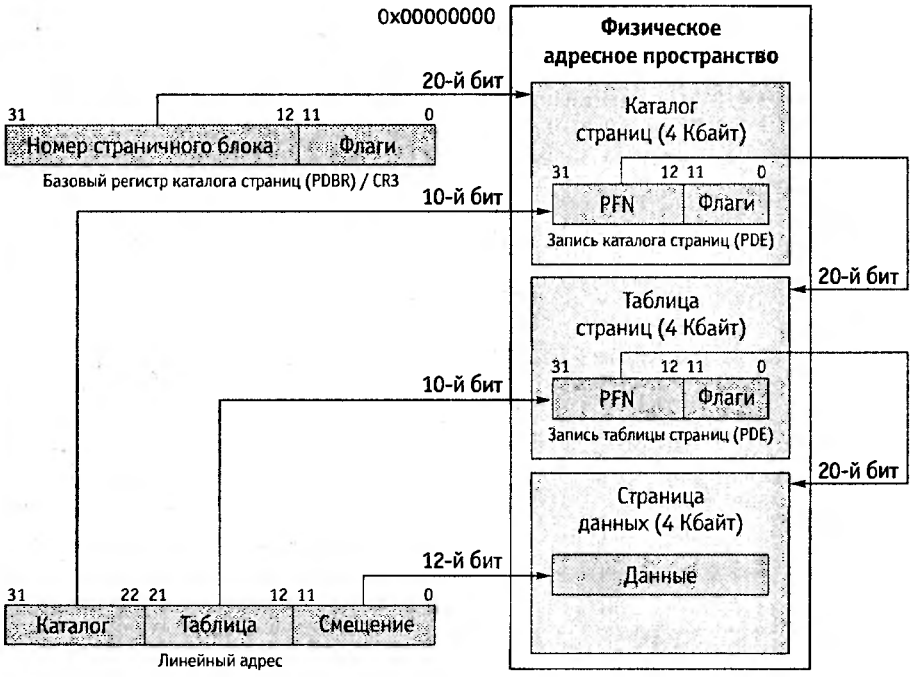

Рис. 4.3. Двухуровневая организация страничной памяти с размером страниц в 4 Кбайт

В схеме страничной памяти с размером страницы 4 Кбайт линейное адресное пространство размером 4 Гбайт адресуется посредством двухуровневого механизма косвенной адресации. В худшем случае потребуется 1 048 576 записей PTE, чему соответствуют 1024 таблицы страниц (количество записей PDE в каталоге страниц). Поскольку каталог страниц и каждая таблица страниц занимает 4 Кбайт,

максимальные накладные расходы памяти в этой модели страничной памяти составят 4 Кбайт плюс 4 Мбайт, или 4100 Кбайт. Это разумная цена за деление всего пространства размером 4 Гбайт на фрагменты в 4 Кбайт, которые можно отобразить на любой линейный адрес.

В режиме страничного обмена с размером страниц в 4 Мбайт все значительно проще, поскольку один уровень косвенной адресации исчезает, как показано на рис. 4.4. Указатель PDBR точно так же указывает на каталог страниц, но теперь используются только старшие 10 бит для выбора PDE и 22 бита смещения. Накладные расходы при такой схеме организации памяти составят не более чем 4 Кбайт, поскольку дополнительная память расходуется лишь для хранения каталога страниц. Этого вполне достаточно для охвата всего адресного пространства в 4 Гбайт. Таким образом, страницы в 4 Мбайт обладают преимуществом низких накладных расходов на управление памятью ценой адресации более крупных блоков памяти.

У обоих режимов страничной организации памяти есть свои преимущества и недостатки. К счастью, проектировщики операционной системы не обязаны строго придерживаться одного из них, а могут использовать процессор, работающий в смешанном режиме. Например, Windows 2000 работает со страницами в 4 Мбайт в диапазоне адресов памяти с 0x80000000 по 0x9FFFFFFF, в котором расположены модули hal.dll и ntoskrnl.exe. Остальные блоки линейных адресов управляются как фрагменты по 4 Кбайт. Эта смешанная архитектура рекомендована Intel для повышения производительности системы, поскольку для кэширования страниц в 4 Кбайт и 4 Мбайт используются различные буферы быстрого преобразования адресов (Translation Lookaside Buffer, TLB) внутри процессоров i386 (Intel, 1999c, pp. 3–22f). Ядро операционной системы, как правило, весьма велико и постоянно находится в памяти, поэтому при хранении его в нескольких страницах размером 4 Кбайт постоянно расходовалось бы дорогостоящее пространство буфера TLB.

Заметьте, что все этапы по преобразованию адресов происходят в физической памяти. В регистре PDBR и во всех записях PDE и PTE содержатся указатели на физические адреса. Единственный линейный адрес, встречающийся на рис. 4.3 и 4.4, — это прямоугольник в левом нижнем углу экрана, задающий адрес, который нужно преобразовать в смещение внутри физической страницы. С другой стороны, приложения должны работать с линейными адресами, игнорируя физические адреса. Этот пробел можно восполнить, отобразив каталог страниц и все связанные с ним таблицы страниц в линейное адресное пространство. В Windows 2000 и Windows NT все записи PDE и PTE доступны в диапазоне адресов с 0xC0000000 по 0xC03FFFFF, представляющем собой линейную область памяти размером 4 Мбайт. Понятно, что это максимальный размер памяти, который может занять уровень таблиц страниц в режиме страничной памяти с размером страниц в 4 Кбайт. Связанную с линейным адресом запись PTE можно найти, просто используя ее старшие 20 бит в качестве индекса в массиве 32-битных записей PTE, начинающихся с адреса 0xC0000000. Например, запись PTE адреса 0x00000000 расположена в 0xC0000000. Индекс PTE адреса 0x80000000 вычисляется путем побитового сдвига вправо на 12 бит для получения старших 20 бит, в результате чего будет получено значение 0x80000. Поскольку каждая запись PTE занимает четыре байта, требуемая PTE находится по адресу 0xC0000000 + (4 * 0x80000) = 0xC0200000. Полученный резуль-

тат весьма интересен — адрес, который делит адресное пространство в 4 Гбайт на две равные части, отображается в адрес PTE, который также делит массив записей PTE на две равные части.

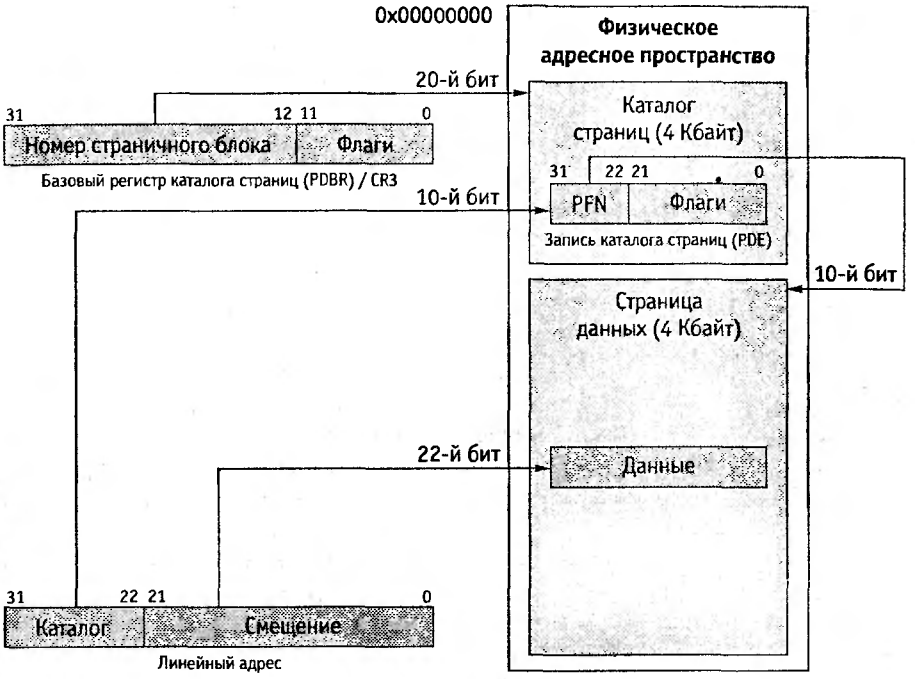

Рис. 4.4. Одноуровневая организация страничной памяти с размером страниц в 4 Мбайт

Теперь давайте сделаем еще один шаг вперед и вычислим полный адрес самого массива PTE. Главная формула отображения адресов следующая: ((LinearAddress >> 12) * 4) + 0xC0000000. При установке значения LinearAddress в 0xC0000000 мы получим 0xC0300000. Приостановимся на мгновение: запись по линейному адресу 0xC0300000 указывает на начало массива записей PTE в физической памяти. Вернемся теперь к рис. 4.3. 1024 записи, расположенные по адресу 0xC0300000, должны быть каталогом страниц! Это особое расположение массивов PDE и PTE используется различными функциями управления памятью, реализованными в ntoskrnl.exe. Например, документированные функции API MmIsAddressValid() и MmGetPhysicalAddress() для 32-битного линейного адреса ищут его PDE и, если это возможно, PTE и исследуют их содержимое. MmIsAddressValid() просто проверяет, присутствует ли в данный момент в физической памяти требуемая страница. Если ответ отрицательный, то линейный адрес либо недопустим, либо ссылается на перемещенную на резервное устройство страницу, представленную рядом системных файлов подкачки (pagefile). Функция MmGetPhysicalAddress() сначала извлекает соответствующий линейному адресу номер страничного блока (PFN), значение которого равно значению базового адреса связанной с ним физической страницы, деленному на размер страницы. Затем функция вычисляет смещение в этой странице, извлекая

последние 12 значащих битов линейного адреса, и добавляет смещение к физическому базовому адресу, определенному в PFN.

При более тщательном исследовании реализации функции MmGetPhysicalAddress() выявляется еще одна интересная особенность схемы размещения памяти Windows 2000. В первую очередь код проверяет, находится ли линейный адрес в диапазоне адресов с 0x80000000 по 0x9FFFFFFF. Как уже говорилось, здесь «прописаны» модули hal.dll и ntoskrnl.exe и в этом блоке адресов Windows 2000 использует страницы объемом 4 Мбайт. Интересно то, что функция MmGetPhysicalAddress() вообще не проверяет записи PDE и PTE, если адрес не принадлежит этому диапазону. Вместо этого она просто устанавливает три старших бита в ноль, добавляет, как обычно, смещение в байтах и возвращает этот результат как физический адрес. А это означает, что диапазон физических адресов с 0x00000000 по 0x1FFFFFFF один к одному отображается в диапазон линейных адресов с 0x80000000 по 9FFFFFFF! Зная, что модуль ntoskrnl.exe всегда загружается по линейному адресу 0x80400000, можно сделать вывод, что ядро Windows 2000 всегда можно найти по физическому адресу 0x00400000, который одновременно является базовым адресом второй страницы в 4 Мбайт в физической памяти. И в самом деле, исследование этих областей памяти подтверждает верность сделанных предположений. Вы сами сможете в этом убедиться при помощи программы просмотра памяти, представленной в этой главе.

Структуры данных

Некоторые части кода примеров в этой главе связаны с низкоуровневым управлением памятью и заглядывают внутрь механизмов, описанных выше. Для упрощения задачи я определил несколько структур данных на C. Поскольку многие элементы данных внутри процессора i386 представляют собой несколько связанных одиночных битов или групп битов, битовые поля C очень удобны. Битовые поля позволяют эффективно обращаться к отдельным битам или извлекать непрерывные группы битов из слов данных большего размера. Компилятор среды Microsoft Visual C/C++ генерирует для операций с битовыми полями весьма эффективный код. В листинге 4.2 приведена первая часть определений типов данных процессора, содержащая следующие типы:

- X86_REGISTER — это базовый беззнаковый 32-битный целочисленный тип для представления различных регистров процессора, в числе которых все регистры общего назначения, регистры указателей и команд, а также индексные, отладочные и проверочные регистры;
- X86_SELECTOR представляет 16-битный селектор сегмента так, как он хранится в сегментных регистрах CS, DS, ES, FS, GS и SS. На рис. 4.1 и 4.2 селекторы изображены как старшая треть логического 48-битного адреса, выполняя функции индекса в таблице дескрипторов. Для удобства вычислений 16-битное значение селектора расширяется до 32 бит, старшая половина которых помечена как «зарезервированные». Обратите внимание на то, что структура данных X86_SELECTOR является объединением двух структур. Первая структура определяет значение селектора как упакованное 16-битное значение типа WORD с именем wValue,

а вторая трактует его значение как набор битовых полей. Поле RPL задает запрашиваемый уровень привилегий (Requested Privilege Level), который в Windows 2000 равен либо 0 (режим ядра), либо 3 (пользовательский режим). Бит TI работает как переключатель между локальной и глобальной таблицами дескрипторов (GDT/LDT);

- X86_DESCRIPTOR определяет формат для записи в таблице, на которую указывает селектор. Это 64-разрядная величина с весьма запутанной структурой, что явилось следствием ее исторического развития. Линейный базовый адрес, определяющий начальное расположение соответствующего сегмента, распределен по трем битовым полям Base1, Base2 и Base3, где Base3 представляет наименее значащую часть. Граница сегмента, определяемая как размер сегмента минус один, разделено на пару Limit1 и Limit2, в которой первое значение представляет менее значащую половину. В оставшихся битовых полях хранятся различные свойства сегмента (Intel, 1999c, pp. 3–11). Например, бит G (granularity) определяет уровень масштабирования сегмента. Если он равен нулю, граница сегмента задается в байтах, если единице, то значение границы нужно умножать на 4 Кбайт. Аналогично X86_SELECTOR, структура данных X86_DESCRIPTOR представляет собой объединение, чтобы можно было по-разному интерпретировать ее значение. Члены альтернативной структуры dValueLow и dValueHigh полезны, если нужно копировать дескрипторы без учета их внутренней структуры;
- X86_GATE выглядит похожей на X86_DESCRIPTOR. В действительности эти структуры данных связаны: X86_DESCRIPTOR является записью в таблице GDT и описывает свойства памяти для сегмента, а структура данных X86_GATE — это запись в таблице дескрипторов прерываний, описывающая свойства памяти для обработчика прерываний. В IDT могут храниться шлюзы (gates) задачи, прерывания и ловушки. (Нет, Bill Gates[1] *не* хранится в IDT!) Структура данных X86_GATE может представлять все три типа шлюзов, что задается битовым полем Type. Тип 5 соответствует шлюзу задачи (task gate), типы 6 и 14 — шлюзам прерываний (interrupt gate), а типы 7 и 15 представляют шлюзы ловушек (trap gate). Самый старший бит типа определяет размер шлюза: у 16-битных шлюзов он установлен в 0, значение 1 задает 32-битный шлюз;
- X86_TABLE представляет собой хитрую структуру, используемую для чтения значений регистров GDTR и IDTR при помощи соответствующих ассемблерных инструкций SGDT (store GDT register) и SIDT (store IDT register), см. Intel, 1999b, pp. 3–636. Обе инструкции ожидают 48-битный операнд, в котором должны храниться значения границы и базового адреса. Для выравнивания по границе DWORD для 32-битного базового адреса структура X86_TABLE начинается с 16-битного пустого члена wReserved. В зависимости от применяемой инструкции (SGDT или SIDT) базовый адрес должен трактоваться как указатель либо на дескриптор, либо на шлюз, что реализовано при помощи объединения типов данных PX86_DESCRIPTOR и PX86_GATES. Член структуры wLimit одинаков для обоих типов.

[1] Игра слов: Bill Gates (Билл Гейтс) и gates (шлюзы). — *Примеч. ред.*

Листинг 4.2. Регистры, селекторы, дескрипторы, шлюзы и таблицы процессора i386

```
//  ============================================================
// СТРУКТУРЫ ДАННЫХ INTEL X86. ЧАСТЬ 1 ИЗ 3
//  ============================================================

typedef DWORD X86_REGISTER, *PX86_REGISTER,
              **PPX86_REGISTER;

//  ------------------------------------------------------------

typedef struct _X86_SELECTOR
      {
      union
           {
           struct
                {
                WORD wValue;                  // упакованное значение
                WORD wReserved;
                };
           struct
                {
                unsigned RPL        :  2;     // запрашиваемый
                                              // уровень привилегий
                unsigned TI         :  1;     // индикатор таблицы:
                                              // 0=GDT, 1=LDT
                unsigned Index      : 13;     // индекс в таблице
                                              // дескрипторов
                unsigned Reserved   : 16;
                };
           };
      }
      X86_SELECTOR, *PX86_SELECTOR, **PPX86_SELECTOR;

#define X86_SELECTOR_ sizeof (X86_SELECTOR)

//  ------------------------------------------------------------

typedef struct _X86_DESCRIPTOR
      {
      union
           {
           struct
                {
                DWORD dValueLow;                    // упакованное значение
                DWORD dValueHigh;
                };
           struct
                {
                unsigned Limit1          : 16;      // биты 15..00
                unsigned Base1           : 16;      // биты 15..00
                unsigned Base2           :  8;      // биты 23..16
                unsigned Type            :  4;      // тип сегмента
                unsigned S               :  1;      // тип (0=система,
                                                    // 1=код/данные)
                unsigned DPL             :  2;      // уровень привилегий
                                                    // дескриптора
```

```
                unsigned P              : 1;    // присутствие
                                                // сегмента
                unsigned Limit2         : 4;    // биты 19..16
                unsigned AVL            : 1;    // в распоряжении
                                                // программиста
                unsigned Reserved       : 1;
                unsigned DB             : 1;    // 0=16-бит, 1=32-бит
                unsigned G              : 1;    // масштаб (1=4KB)
                unsigned Base3          : 8;    // биты 31..24
                };
            };
        }
        X86_DESCRIPTOR, *PX86_DESCRIPTOR, **PPX86_DESCRIPTOR;

#define X86_DESCRIPTOR_ sizeof (X86_DESCRIPTOR)

//  ------------------------------------------------------

typedef struct _X86_GATE
        {
        union
            {
            struct
                {
                DWORD dValueLow;            // упакованное значение
                DWORD dValueHigh;
                };
            struct
                {
                unsigned Offset1        : 16;   // биты 15..00
                unsigned Selector       : 16;   // селектор сегмента
                unsigned Parameters     :  5;   // параметры
                unsigned Reserved       :  3;
                unsigned Type           :  4;   // тип и размер шлюза
                unsigned S              :  1;   // всегда 0
                unsigned DPL            :  2;   // уровень привилегий
                                                // дескриптора
                unsigned P              :  1;   // присутствие
                                                // сегмента
                unsigned Offset2        : 16;   // биты 31..16
                };
            };
        }
        X86_GATE, *PX86_GATE, **PPX86_GATE;

#define X86_GATE_ sizeof (X86_GATE)

//  ------------------------------------------------------

typedef struct _X86_TABLE
        {
        WORD wReserved;                         // явное выравнивание
                                                // по границе
                                                // 32-битного слова
        WORD wLimit;                            // размер таблицы
```

продолжение

Листинг 4.2 (*продолжение*)

```
    union
        {
        PX86_DESCRIPTOR     pDescriptors:       // для инстр. sgdt
        PX86_GATE           pGates:             // для инстр. sidt
        };
    }
    X86_TABLE. *PX86_TABLE. **PPX86_TABLE;

#define X86_TABLE_ sizeof (X86_TABLE)

//  ================================================
```

Следующий набор структур управления памятью i386, представленный в листинге 4.3, относится к подкачке страниц по запросу и содержит несколько элементов, проиллюстрированных на рис. 4.3 и 4.4:

- X86_PDBR — это, безусловно, структурное представление регистра процессора CR3, также известного как базовый регистр каталога страниц (Page Directory Base Register, PDBR). В старших 20 битах расположен номер страничного блока (PFN), индекс в массиве физических страниц объемом 4 Кбайт. Значение PFN=0 соответствует физическому адресу 0x00000000, PFN=1 — адресу 0x00001000 и т. д. Двадцати бит как раз достаточно, чтобы охватить все 4 Гбайт адресного пространства. Значение PFN в регистре PDBR — это индекс физической страницы, в которой хранится каталог страниц. Большая часть оставшихся бит зарезервированы, за исключением бита № 3[1], управляющего постраничной сквозной записью (page level write through, PWT), и бита № 4, в установленном состоянии отключающего постраничное кэширование;
- X86_PDE_4M и X86_PDE_4K представляют собой альтернативные варианты записей в каталоге страниц (PDE) для страниц объемом 4 Мбайт и 4 Кбайт. В каталоге страниц может содержаться максимум 1024 записи PDE. Здесь PFN также определяет номер страничного блока (page frame number), указывая на страницу более низкого уровня. Для записи PDE для страниц размером 4 Мбайт размер битового поля PFN только 10 бит, адресующих страницу данных объемом 4 Мбайт. 20-битное поле PFN для PDE страниц в 4Кбайт указывает на таблицу страниц, з которой окончательно выбирается страница физических данных. Оставшиеся биты определяют разнообразные свойства, самые интересные из которых — это бит размера страницы PS («Page Size»), управляющий размером страницы (0 соответствует 4 Кбайт, 1 — 4 Мбайт), и бит наличия страницы P («Present»), указывающий на наличие соответствующей страницы данных (в режиме 4 Мбайт) или таблицы страниц (в режиме 4 Кбайт);
- X86_PTE_4K определяет внутреннюю структуру записи в таблице страниц (page table entry, PTE). Так же как и каталог страниц, в таблице страниц может храниться до 1024 записей. Структуры данных X86_PTE_4K и X86_PDE_4K отличаются только тем, что в первой нет бита PS, в котором нет необходимости, поскольку размер страницы должен быть 4 Кбайт, как определено битом PS записей

[1] Именно бита номер 3, то есть четвертого бита (начиная с нулевого в битовом поле). — *Примеч. перев.*

PDE. Отметьте, что нет такого понятия, как PTE для страниц объемом 4 Мбайт, поскольку в модели памяти для страниц размером 4 Мбайт не требуется промежуточный уровень таблицы страниц;

- X86_PNPE представляет «запись для отсутствующей страницы» (page not present entry, PNPE), то есть запись PDE или PTE с нулевым битом P. В соответствии со справочными руководствами Intel остальные 31 бит «доступны операционной системе или исполняющей системе» (Intel, 1999c, pp. 3–28). Если линейный адрес отображается в PNPE, это означает, что либо этот адрес не используется, либо что он указывает на страницу, в данный момент выгруженную в один из файлов подкачки. Windows 2000 использует 31 свободный бит PNPE для хранения информации о состоянии страницы. Структура этой информации недокументирована, но похоже, что бит № 10, названный PageFile в листинге 4.3, устанавливается при выгрузке страницы. В этом случае в битовых полях Reserved1 и Reserved2 содержатся значения, позволяющие системе находить страницу в файлах подкачки, чтобы сразу загрузить страницу, как только один из ее линейных адресов будет задействован в инструкции чтения или записи памяти;
- X86_PE включена для удобства. Это просто объединение всех возможных форм записей страниц, в которое включены содержимое PDBR, записи PDE для страниц объемом 4 Кбайт и 4 Мбайт, а также записи PTE и PNPE.

Листинг 4.3. Значения i386 PDBR, PDE, PTE и PNPE

```
//  ================================================
// СТРУКТУРЫ ДАННЫХ INTEL X86. ЧАСТЬ 2 ИЗ 3
//  ================================================

// базовый регистр каталога страниц (cr3)
typedef struct _X86_PDBR
      {
      union
          {
          struct
               {
               DWORD dValue;                    // упакованное значение
               };
          struct
               {
               unsigned Reserved1       : 3;
               unsigned PWT             : 1;    // постраничная
                                                // сквозная запись
               unsigned PCD             : 1;    // постраничное
                                                // кэширование
                                                // отключено
               unsigned Reserved2       : 7;
               unsigned PFN             : 20;   // номер страничного
                                                // блока
               };
          };
      }
      X86_PDBR, *PX86_PDBR, **PPX86_PDBR;

#define X86_PDBR_ sizeof (X86_PDBR)
```

продолжение

Листинг 4.3 (*продолжение*)

```
//  ---------------------------------------------------------

// запись в каталоге страниц (размер страницы 4-Мбайт)
typedef struct _X86_PDE_4M
      {
      union
           {
           struct
                {
                DWORD dValue;                    // упакованное значение
                };
           struct
                {
                unsigned P              : 1;     // наличие страницы в
                                                 // основной памяти
                                                 // (1 = присутствует)
                unsigned RW             : 1;     // чтение/запись
                unsigned US             : 1;     // пользователь/
                                                 // супервизор
                unsigned PWT            : 1;     // постраничная
                                                 // сквозная запись
                unsigned PCD            : 1;     // постраничное
                                                 // кэширование
                                                 // отключено
                unsigned A              : 1;     // к странице было
                                                 // обращение
                unsigned D              : 1;     // страница изменена
                unsigned PS             : 1;     // размер страницы
                                                 // (1 = 4-Мбайт)
                unsigned G              : 1;     // глобальная
                                                 // страница
                unsigned Available      : 3;     // в распоряжении
                                                 // программиста
                unsigned Reserved       : 10;
                unsigned PFN            : 10;    // номер страничного
                                                 // блока
                };
           };
      }
      X86_PDE_4M, *PX86_PDE_4M, **PPX86_PDE_4M;

#define X86_PDE_4M_ sizeof (X86_PDE_4M)

//  ---------------------------------------------------------

// запись в каталоге страниц (размер страницы 4 Кбайт)
typedef struct _X86_PDE_4K
      {
      union
           {
           struct
                {
                DWORD dValue;                    // упакованное значение
                };
           struct
                {
```

```
            unsigned P              : 1;    // наличие страницы в
                                            // основной памяти
                                            // (1 = присутствует)
            unsigned RW             : 1;    // чтение/запись
            unsigned US             : 1;    // пользователь/
                                            // супервизор
            unsigned PWT            : 1;    // постраничная
                                            // сквозная запись
            unsigned PCD            : 1;    // постраничное
                                            // кэширование
                                            // отключено
            unsigned A              : 1;    // к странице было
                                            // обращение
            unsigned Reserved       : 1;    // страница изменена
            unsigned PS             : 1;    // размер страницы
                                            // (0 = страница
                                            // объемом 4-Кбайт)
            unsigned G              : 1;    // глобальная
                                            // страница
            unsigned Available      : 3;    // в распоряжении
                                            // программиста
            unsigned PFN            : 20;   // номер страничного
                                            // блока
            };
        };
    }
    X86_PDE_4K, *PX86_PDE_4K, **PPX86_PDE_4K;

#define X86_PDE_4K_ sizeof (X86_PDE_4K)

//  ------------------------------------------------------

// запись в таблице страниц (размер страницы 4 Кбайт)
typedef struct _X86_PTE_4K
    {
    union
        {
        struct
            {
            DWORD dValue;                   // упакованное значение
            };
        struct
            {
            unsigned P              : 1;    // наличие страницы в
                                            // основной памяти
                                            // (1 = присутствует)
            unsigned RW             : 1;    // чтение/запись
            unsigned US             : 1;    // пользователь/
                                            // супервизор
            unsigned PWT            : 1;    // постраничная
                                            // сквозная запись
            unsigned PCD            : 1;    // постраничное
                                            // кэширование
                                            // отключено
            unsigned A              : 1;    // к странице было
                                            // обращение
```

продолжение

Листинг 4.3 (*продолжение*)

```
            unsigned D              : 1;    // страница изменена
            unsigned Reserved       : 1;
            unsigned G              : 1;    // глобальная
                                            // страница
            unsigned Available      : 3;    // в распоряжении
                                            // программиста
            unsigned PFN            : 20;   // номер страничного
                                            // блока
            };
        };
    }
    X86_PTE_4K, *PX86_PTE_4K, **PPX86_PTE_4K;

#define X86_PTE_4K_ sizeof (X86_PTE_4K)

//  ------------------------------------------------------

    // запись для отсутствующей страницы
typedef struct _X86_PNPE
    {
    union
        {
        struct
            {
            DWORD dValue:                   // упакованное значение
            };
        struct
            {
            unsigned P              : 1;    // наличие страницы в
                                            // основной памяти
                                            // (0 = отсутствует)
            unsigned Reserved1      : 9;
            unsigned PageFile       : 1;    // страница выгружена
                                            // в файл подкачки
            unsigned Reserved2      : 21;
            };
        };
    }
    X86_PNPE, *PX86_PNPE, **PPX86_PNPE;

#define X86_PNPE_ sizeof (X86_PNPE)

//  ------------------------------------------------------

typedef struct _X86_PE                  // обобщенная запись страницы
    {
    union
        {
        DWORD           dValue;     // упакованное значение
        X86_PDBR        pdbr;       // базовый регистр каталога
                                    // страниц
        XB6_PDE_4M      pde4M;      // запись в каталоге страниц
                                    // (размер страницы 4 Мбайт)
        XB6_PDE_4K      pde4K;      // запись в каталоге страниц
                                    // (размер страницы 4 Кбайт)
```

```
        X86_PTE_4K          pte4K:    // запись в таблице страниц
                                      // (размер страницы 4 Кбайт)
        X86_PNPE            pnpe:     // запись для отсутствующей
                                      // страницы
        };
    }
    X86_PE. *PX86_PE. **PPX86_PE;

#define X86_PE_ sizeof (X86_PE)

//  ================================================
```

В листинге 4.4 приведено представление линейных адресов в виде структур, представляющих собой формальные определения прямоугольников «линейный адрес» на рис. 4.3 и 4.4:

- X86_LINEAR_4M представляет формат линейных адресов, ссылающихся на страницу данных объемом 4 Мбайт, как показано на рис. 4.4. Индекс каталога страниц PDT выбирает одну из записей PDE в каталоге, на который в данный момент указывает регистр PDBR. 22-битный член структуры Offset указывает на требуемый адрес внутри соответствующей физической страницы объемом 4 Мбайт;
- X86_LINEAR_4K — это вариант линейного адреса для режима 4 Кбайт. Как показано на рис. 4.3, он состоит из трех битовых полей. Как и адрес в странице объемом 4 Мбайт, старшие 10 бит PDI выбирают запись PDE. Индекс в таблице страниц PTI выполняет схожие задачи, указывая на PTE внутри таблицы страниц, адресуемой этой записью PDE. Оставшиеся 12 бит задают смещение в конечной физической странице размером 4 Кбайт.
- X86_LINEAR — еще одна определенная для удобства структура, просто объединяющая типы X86_LINEAR_4M и X86_LINEAR_4K в одном типе данных.

Макроопределения и константы

Определения из листинга 4.5 дополняют структуры из листингов 4.2–4.4 для облегчения работы с механизмами управления памятью i386. Их можно отнести в три основные группы. Первая группа предназначена для работы с линейными адресами:

1. X86_PAGE_MASK, X86_PDI_MASK и X86_PTI_MASK представляют собой битовые маски, делящие линейные адреса на составляющие части. Они основаны на константах PAGE_SHIFT (12), PDI-SHIFT (22) и PTI_SHIFT (12), определенных в заголовочном файле ntddk.h комплекта Windows 2000 DDK. Численное значение маски X86_PAGE_MASK равно 0xFFFFF000, что сбрасывает биты линейного адреса, соответствующие части смещения в странице 4 Кбайт (см. X86_LINEAR_4K), маска X86_PDI_MASK эквивалентна 0xFFC00000 и «вырезает» из линейного адреса 10 старших битов индекса PDI (см. X86_LINEAR_4M и X86_LINEAR_4K). X86_PTI_MASK равна 0x003FF0000 и сбрасывает в линейном адресе все биты, кроме битов индекса таблицы страниц PTI (см. X86_LINEAR_4K).

Листинг 4.4. Линейные адреса i386

```
//  ================================================
// СТРУКТУРЫ ДАННЫХ INTEL X86. ЧАСТЬ 3 ИЗ 3
//  ================================================

// линейный адрес (размер страницы 4 Мбайт)
typedef struct _X86_LINEAR_4M
      {
      union
           {
           struct
                {
                PVDID pAddress;              // упакованный адрес
                };
           struct
                {
                unsigned Dffset     : 22;    // смещение в странице
                unsigned PDI        : 10;    // индекс в таблице
                                             // каталогов
                };
           };
      }
      X86_LINEAR_4M, *PX86_LINEAR_4M, **PPX86_LINEAR_4M;

#define X86_LINEAR_4M_ sizeof (X86_LINEAR_4M)

//  ------------------------------------------------------

// линейный адрес (размер страницы 4 Кбайт)
typedef struct _X86_LINEAR_4K
      {
      union
           {
           struct
                {
                PVOID pAddress;              // упакованный адрес
                };
           struct
                {
                unsigned Dffset     : 12;    // смещение в странице
                unsigned PTI        : 10;    // индекс в таблице
                                             // страниц
                unsigned PDI        : 10;    // индекс в таблице
                                             // каталогов
                };
           };
      }
      X86_LINEAR_4K, *PX86_LINEAR_4K, **PPX86_LINEAR_4K;

#define XB6_LINEAR_4K_ sizeof (X86_LINEAR_4K)

//  ------------------------------------------------------

typedef struct _X86_LINEAR                   // общий линейный адрес
      {
      union
           {
```

```
        PVOID           pAddress;       // упакованный адрес
        X86_LINEAR_4M   linear4M;       // линейный адрес
                                        // (размер страницы 4 Мбайт)
        X86_LINEAR_4K   linear4K;       // линейный адрес
                                        // (размер страницы 4 Кбайт)
        };
    }
    X86_LINEAR, *PX86_LINEAR, **PPX86_LINEAR;

#define XB6_LINEAR_ sizeof (X86_LINEAR)

//  ================================================
```

2. X86_PAGE(), X86_PDI() и X86_PTI() используют определенные выше константы для вычисления индекса страницы, PDI или PTI, заданного линейного адреса. X86_PAGE() обычно используется для чтения записи PTE из массива PTE Windows 2000, начинающегося по адресу 0xC0000000, X86_PDI() и X86_PTI() накладывают маску X86_PDI_MASK или X86_PTI_MASK на аргумент-указатель и сдвигают полученный индекс вправо на соответствующее число разрядов.
3. X86_OFFSET_4M() и X86_OFFSET_4K() извлекают часть смещения из линейных адресов 4 Мбайт и 4 Кбайт.
4. X86_PAGE_4M и X86_PAGE_4K вычисляют размеры страниц объемом 4 Мбайт и 4 Кбайт при помощи определенных в DDK констант PDI_SHIFT и PTI_SHIFT, получая значения X86_PAGE_4M = 4 194 304 и X86_PAGE_4K = 4096. Заметьте, что X86_PAGE_4K эквивалентна константе DDK PAGE_SIZE, также определенной в ntddk.h.
5. X86_PAGES_4M и X86_PAGES_4K содержат число страниц размером 4 Мбайт и 4 Кбайт, помещающихся в линейном адресном пространстве 4 Гбайт. X86_PAGES_4M равно 1024, X86_PAGES_4K — 1 048 576.

Вторая группа макроопределений и констант относится к массивам PDE и PTE Windows 2000. В отличие от нескольких других системных адресов базовые адреса этих массивов не реализованы как глобальные переменные, устанавливаемые на этапе загрузки, а определены как константы. Это легко можно проверить, дизассемблировав функции API диспетчера памяти MmGetPhysicalAddress() или MmIsAddressValid(), в котором эти адреса присутствуют как «загадочные числа». Эти константы не включены в заголовочные файлы DDK, но в листинге 4.5 показан предположительный вариант их определения.

- X86_PAGES — это жестко заданный адрес, который, конечно же, указывает на 0xC0000000, где начинается массив PTE Windows 2000;
- X86_PTE_ARRAY равен X86_PAGES, но приводит значение к типу PX86_PE, то есть указателю на массив структур записей страниц X86_PE, в соответствии с определением из листинга 4.2;
- X86_PDE_ARRAY — довольно хитрое определение, вычисляющее базовый адрес массива PDE на основе расположения массива PTE при помощи константы PTI_SHIFT. Как объяснялось раньше, для отображения линейного адреса в адрес PTE используется общая формула ((LinearAddress >> 12) * 4) + 0xC0000000, а к каталогу страниц можно обратиться, установив LinearAddress в 0xC0000000. Больше определение X86_PDE_ARRAY ничего не делает.

Листинг 4.5. Дополнительные определения для управления памятью i386

```
//  ================================================================
// МАКРООПРЕДЕЛЕНИЯ И КОНСТАНТЫ INTEL X86
//  ================================================================

#define X86_PAGE_MASK     (0 - (1 << PAGE_SHIFT))
#define X86_PAGE(_p)      (((DWORD) (_p) & X86_PAGE_MASK) >> PAGE_SHIFT)

#define X86_PDI_MASK (0 - (1 << PDI_SHIFT))
#define X86_PDI(_p)  (((DWORD) (_p) & X86_PDI_MASK) >> PDI_SHIFT)

#define X86_PTI_MASK      ((0 - (1 << PTI_SHIFT)) & ~X86_PDI_MASK)
#define X86_PTI(_p)       (((DWORD) (_p) & X86_PTI_MASK) >> PTI_SHIFT)

#define X86_OFFSET_4M(_p)[1]     ((_p) & ~(X86_PDI_MASK  ))
#define X86_OFFSET_4K(_p) ((_p) & ~(X86_PDI_MASK | X86_PTI_MASK))

#define X86_PAGE_4M       (1 << PDI_SHIFT)
#define X86_PAGE_4K       (1 << PTI_SHIFT)

#define X86_PAGES_4M      (1 << (32 - PDI_SHIFT))
#define X86_PAGES_4K      (1 << (32 - PTI_SHIFT))

//  ----------------------------------------------------------

#define X86_PAGES              0xC0000000
#define X86_PTE_ARRAY          ((PX86_PE) X86_PAGES)
#define X86_PDE_ARRAY          (X86_PTE_ARRAY + (X86_PAGES >> PTI_SHIFT))

//  ----------------------------------------------------------

#define X86_SELECTOR_RPL                0x0003
#define X86_SELECTOR_TI                 0x0004
#define X86_SELECTOR_INDEX              0xFFF8
#define X86_SELECTDR_SHIFT              3

#define X86_SELECTOR_LIMIT         (X86_SELECTDR_INDEX >> X86_SELECTOR_SHIFT)

//  ----------------------------------------------------------

#define  X86_DESCRIPTOR_SYS_TSS16A          0x1
#define  X86_DESCRIPTOR_SYS_LDT             0x2
#define  X86_DESCRIPTOR_SYS_TSS16B          0x3
#define  X86_DESCRIPTOR_SYS_CALL16          0x4
#define  X86_DESCRIPTOR_SYS_TASK            0x5
#define  X86_DESCRIPTOR_SYS_INT16           0x6
#define  X86_DESCRIPTOR_SYS_TRAP16          0x7
#define  X86_DESCRIPTOR_SYS_TSS32A          0x9
#define  X86_DESCRIPTOR_SYS_TSS32B          0xB
#define  X86_DESCRIPTOR_SYS_CALL32          0xC
#define  X86_DESCRIPTOR_SYS_INT32           0xE
#define  X86_DESCRIPTOR_SYS_TRAP32          0xF
```

[1] На компакт-диске перед этим определен еще дополнительный макрос #define X86_OFFSET(_p,_m) ((DWORD_PTR) (_p) & ~(_m)), и эти два последующих макроса определены через него, но это эквивалентно. — *Примеч. перев.*

```
//  ------------------------------------------------------------

#define  X86_DESCRIPTOR_APP_ACCESSED          0x1
#define  X86_DESCRIPTOR_APP_READ_WRITE        0x2
#define  X86_DESCRIPTOR_APP_EXECUTE_READ      0x2
#define  X86_DESCRIPTOR_APP_EXPAND_DOWN       0x4
#define  X86_DESCRIPTOR_APP_CONFORMING        0x4
#define  X86_DESCRIPTOR_APP_CODE              0x8

//  ============================================================
```

В последних двух секциях листинга 4.5 описаны константы для оперирования селекторами и специальными типами дескрипторов, дополняющие определения из листинга 4.2.

- X86_SELECTOR_RPL, X86_SELECTOR_TI и X86_SELECTOR_INDEX — это битовые маски, соответствующие членам структуры X86_SELECTOR, определенной в листинге 4.2;
- коэффициент сдвига вправо X86_SELECTOR_SHIFT выравнивает вправо значение члена структуры Index селектора;
- X86_SELECTOR_LIMIT задает максимальное значение индекса в селекторе и равно 8191. Это значение определяет максимальный размер таблицы дескрипторов. Каждый индекс селектора указывает на дескриптор, состоящий из 64 бит, или 8 байт (см. X86_DESCRIPTOR в листинге 4.2), следовательно, максимальный размер таблицы дескрипторов равен 8192 * 8 = 64 Кбайт;
- список констант X86_DESCRIPTOR_SYS_* определяет возможные значения члена структуры Type дескриптора, если его бит S равен нулю, что является признаком системного дескриптора. Схема битовых полей дескриптора определена в листинге 4.2 в структуре X86_DESCRIPTOR. Типы системных дескрипторов подробно описаны в руководствах Intel (Intel, 1999c, pp. 3–15f) и сведены вместе в табл. 4.1.

  Константы X86_DESCRIPTOR_APP_* из листинга 4.5 определяют диапазон значений члена структуры Type дескриптора, если он выступает в качестве дескриптора приложений и ссылается на сегмент кода или данных, что задается отличным от нуля значением бита S. Поскольку типы дескрипторов приложений могут быть охарактеризованы независимыми свойствами, отраженными в четырех битах типа, константы X86_DESCRIPTOR_APP_* определены как однобитовые маски, в которых некоторые биты интерпретируются по-разному для сегментов кода и данных:

- X86_DESCRIPTOR_APP_ACCESSED установлен, если к сегменту было обращение;
- X86_DESCRIPTOR_APP_READ_WRITE определяет, возможен ли к сегменту доступ только на чтение или на чтение и запись;
- X86_DESCRIPTOR_APP_EXECUTE_READ определяет, возможен ли к сегменту доступ только на выполнение или на выполнение и чтение;
- X86_DESCRIPTOR_APP_EXPAND_DOWN установлен для сегментов данных, растущих вниз, это свойство, как правило, установлено у сементов стека;

- X86_DESCRIPTOR_APP_CONFORMING определяет, является ли сегмент кода согласующимся (conforming), то есть возможно ли обращение к этому сегменту из кода с меньшим уровнем привилегий (см. Intel, 1999c, pp. 4–13ff);
- X86_DESCRIPTOR_APP_CDDE разделяет сегменты кода и данных. Обратите внимание на то, что сегменты стека принадлежат к категории сегментов данных и запись в них должна быть всегда разрешена.

Обзор системных дескрипторов будет сделан несколько позже, когда можно будет воспользоваться приложением просмотра памяти, представленным в следующих секциях. Таблица 4.1 также представляет собой короткое введение в принципы управления памятью i386. За более подробной информацией по этой теме обратитесь к оригинальным руководствам Intel по процессору Pentium (Intel, 1999a, 1999b, 1999c) или к одному из источников для дополнительного чтения, такому как справочник Роберта Л. Хаммела (Robert L. Hammel) по процессору 80486 (Hummel, 1992).

Таблица 4.1. Типы системных дескрипторов

Имя	Значение	Описание
X86_DESCRIPTOR_SYS_TSS16A	0x1	16-битный доступный (Available) сегмент состояния задачи (Task State Segment)
X86_DESCRIPTOR_SYS_LDT	0x2	Локальная таблица дескрипторов
X86_DESCRIPTOR_SYS_TSS16B	0x3	16-битный занятый (Busy) сегмент состояния задачи
X86_DESCRIPTOR_SYS_CALL16	0x4	16-битный шлюз вызова (Call Gate)
X86_DESCRIPTOR_SYS_TASK	0x5	шлюз задачи (Task Gate)
X86_DESCRIPTOR_SYS_INT16	0x6	16-битный шлюз прерывания (Interrupt Gate)
X86_DESCRIPTOR_SYS_TRAP16	0x7	16-битный шлюз ловушки (Trap Gate)
X86_DESCRIPTOR_SYS_TSS32A	0x9	32-битный доступный (Available) сегмент состояния задачи
X86_DESCRIPTOR_SYS_TSS32B	0xB	32-битный занятый (Busy) сегмент состояния задачи
X86_DESCRIPTOR_SYS_CALL32	0xC	32-битный шлюз вызова
X86_DESCRIPTOR_SYS_INT32	0xE	32-битный шлюз прерывания
X86_DESCRIPTOR_SYS_TRAP32	0xF	32-битный шлюз ловушки

Пример программы просмотра памяти

Microsoft постоянно повторяет, что Windows NT и 2000 — *безопасные* операционные системы. Помимо обеспечения аутентификации пользователей в сетевой среде это утверждение подразумевает также устойчивость к ошибочным приложениям, которые могут нарушить целостность системы, неверно используя указатели или записывая данные за границами структуры данных в памяти. Это всегда представляло собой ужасную проблему в Windows 3.x, в которой операционная система и все приложения располагались в одной области памяти. В Windows NT появилось четкое разделение между памятью системы и приложений, а также между памятью параллельных процессов. Каждому процессу выделяется его собственное адресное пространство объемом в 4 Гбайт, как показано на рис. 4.2. При переключении

задачи сразу же происходит переключение текущего адресного пространства, определяемого значением сегментных регистров, таблиц страниц и других данных управления памятью, которые устанавливаются в значения, требуемые для данного процесса. Такая архитектура не дает приложениям непреднамеренно испортить память других приложений. Каждому процессу необходим также доступ к системным ресурсам, поэтому в пространстве объемом 4 Гбайт всегда находится определенная часть системного кода и данных. Для защиты этих областей памяти от перезаписи кодом чужих приложений применяется другая техника.

Сегментация памяти Windows 2000

Windows 2000 унаследовала базовую схему сегментации памяти от Windows NT 4.0, в которой деление 4 Гбайт адресного пространства процесса на две равные части задано по умолчанию. В меньшей половине, занимающей диапазон адресов 0x00000000–0xFFFFFFFF, содержатся данные приложений и код, выполняющийся в пользовательском режиме, что эквивалентно третьему уровню привилегий (Privilege Level 3) или «третьему кольцу» защиты (Ring 3) в терминологии Intel (Intel, 1999a, pp. 4-8ff; Intel, 1999c, pp. 4–8ff). Верхняя половина, расположенная в диапазоне 0x80000000–0xFFFFFFFF, зарезервирована для операционной системы, выполняющейся в режиме ядра, также известного как нулевой уровень привилегий или нулевое кольцо защиты (Ring 0) Intel. Уровень привилегий определяет возможность исполнения операций и области памяти, к которым может обратиться код. Главным образом это означает, что определенные инструкции процессора запрещены и определенные области памяти недоступны для кода с более низким уровнем привилегий. Например, если приложение пользовательского режима обратится по одному из адресов в верхней половине 4 Гбайт адресного пространства, система возбудит исключение и завершит этот процесс приложения, не оставляя ему возможности для еще одной попытки.

Рисунок 4.5 демонстрирует, что произойдет, если приложение попытается прочитать значение по адресу 0x80000000. Такое жесткое ограничение доступа хорошо для целостности системы, но плохо для инструментов отладки, которые должны быть способны отображать содержимое всех допустимых областей памяти. К счастью, существует простой обходной путь: как и сама система, драйверы режима ядра выполняются на самом высоком уровне привилегий, следовательно, им разрешено выполнять все инструкции процессора и просматривать все области памяти. Технически прием состоит в том, чтобы внедрить в систему драйвер слежения, который считывал бы память в запрошенном диапазоне и отправлял бы ее содержимое вспомогательному приложению, работающему в режиме пользователя. Конечно, даже драйвер режима ядра не может читать значения по адресам виртуальной памяти, которым не соответствует физическая память или память в файле подкачки. Следовательно, такой драйвер должен тщательно проверять все адреса перед обращением к ним, чтобы избежать ужасного синего экрана смерти (Blue Screen Of Death, BSOD). В отличие от исключения приложений, завершающего работу одного только проблемного приложения, исключение драйвера остановит всю систему и потребует перезагрузки.

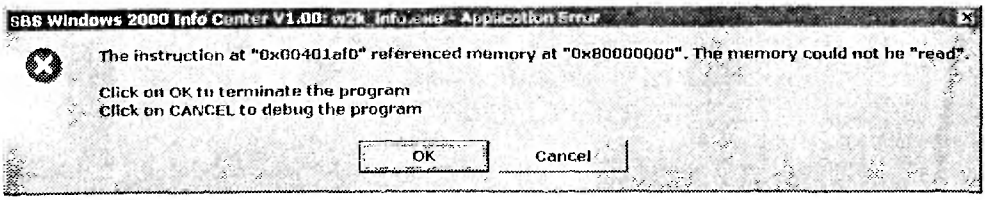

Рис. 4.5. Адреса памяти, начиная с 0x80000000, недоступны в пользовательском режиме

Диспетчер механизма управления вводом-выводом

На прилагающемся к книге компакт-диске в каталоге \src\w2k_spy находятся файлы с исходным кодом универсального устройства слежения (spy device), реализованного в виде драйвера режима ядра. Этот модуль построен на основе шаблона, созданного мастером драйверов из главы 3. Взаимодействие модуля w2k_spy.sys с пользовательским режимом основано на механизме управления вводом-выводом подсистемы Win32 (Device I/O Control, IOCTL), кратко описанной в той же главе. Драйвер слежения определяет устройство по имени \Device\w2k_spy и символьную ссылку \DosDevices\w2k_spy, что необходимо для обращения к устройству из пользовательского режима. Забавно, что пространство имен для символьных ссылок называется \DosDevices. Конечно, мы не будем работать с драйверами DOS. У этого имени глубокие исторические корни, и сейчас оно уже увековечено. После установки символьной ссылки драйвер можно открыть из любого модуля пользовательского режима при помощи стандартной функции Win32 API CreateFile(), задав в качестве пути \\.\w2k_spy. Последовательность символов \\.\ — общая эскейп-последовательность для локальных устройств. Например, \\.\C: обозначает жесткий диск C: в локальной системе. Подробности можно найти в документации для функции CreateFile() в Platform SDK.

Заголовочный файл драйвера w2k_spy.h частично уже приводился выше в листингах 4.2–4.5. Этот файл отчасти похож на заголовочный файл библиотеки DLL: в нем содержатся определения, необходимые самому модулю во время компиляции, но также и достаточное количество информации для взаимодействующего с ним клиентского приложения. Как драйвер (или DLL), так и клиентское приложение включают один и тот же заголовочный файл, и каждый модуль получает все необходимые для работы определения. Однако такая двуликая природа заголовочного файла создает гораздо больше проблем для драйвера режима ядра, чем для DLL, из-за специальных переменных окружения разработки драйверов, определенных Microsoft. К сожалению, заголовочные файлы из комплекта DDK несовместимы с файлами Win32 из комплекта Platform SDK. Нельзя использовать оба заголовочных файла, по крайней мере в проектах на языке C. Это может привести к взаимным блокировкам, когда драйвер режима ядра обращается к константам, макросам и типам данных, недоступным клиентскому приложению, и наоборот. Чтобы избежать таких ситуаций, в файле w2k_spy.c определяется константа-флаг _W2K_SPY_SYS_. В файле w2k_spy.h при помощи инструкций препроцессора #ifdef...#else #endif проверяется факт определения этой константы и соответственно

включаются элементы, отсутствующие в том или ином блоке определений. То есть все определения в ветвях #ifdef _W2K_SPY_SYS_ «видны» только коду драйвера, а определения в ветви #else относятся исключительно к клиентскому приложению. Все части файла w2k_spy.h вне этих условных инструкций относятся к обоим модулям.

При обсуждении мастера драйверов в главе 3 был показан код шаблона драйвера, генерируемого мастером из листинга 3.3. Отправной точкой любого нового проекта на основе этого шаблона обычно является работающая как диспетчер функция DeviceDispatcher(), в качестве аргументов получающая указатель на контекст устройства и указатель на пакет запроса ввода-вывода (I/O Request Packet, IRP). В шаблоне уже обрабатываются основные запросы ввода-вывода IRP_MJ_CREATE, IRP_MJ_CLEANUP и IRP_MJ_CLOSE, отправляемые устройству при его открытии или закрытии клиентом. Функция DeviceDispatcher() возвращает для этих запросов код STATUS_SUCCESS, так что устройство можно открывать или закрывать без возникновения ошибки. Одним устройствам достаточно такой схемы, но другим требуется здесь специальный более или менее сложный код инициализации и очистки. Для всех остальных запросов возвращается код STATUS_NOT_IMPLEMENTED. Расширение кода начинается с изменения этой схемы работы по умолчанию так, чтобы реализовать обработку большего числа запросов. Как уже говорилось, одна из главных задач модуля w2k_spy.sys — отправлять недоступные в режиме пользователя данные приложению Win32 посредством вызовов IOCTL, поэтому мы начнем с добавления в функцию DeviceDispatcher() ветви IRP_MJ_DEVICE_CONTROL. В листинге 4.6 показан обновленный код, так, как он записан в файле w2k_spy.c.

Листинг 4.6. Ветвь IRP_MJ_DEVICE_COMTROL, добавленная в диспетчер

```
NTSTATUS DeviceDispatcher (PDEVICE_CONTEXT   pDeviceContext,
                           PIRP              pIrp)
    {
    PIO_STACK_LOCATION pisl;
    DWORD              dInfo     = 0;
    NTSTATUS           ns        = STATUS_NOT_IMPLEMENTED;

    pisl = IoGetCurrentIrpStackLocation (pIrp);

    switch (pisl->MajorFunction)
        {
        case IRP_MJ_CREATE:
        case IRP_MJ_CLEANUP:
        case IRP_MJ_CLOSE:
            {
            ns = STATUS_SUCCESS;
            break;
            }
        case IRP_MJ_DEVICE_CONTROL:
            {
            ns = SpyDispatcher (pDeviceContext,
                                pisl->Parameters.DeviceIoControl
                                                .IoControlCode,
                                pIrp->AssociatedIrp.SystemBuffer,
```

продолжение ⇩

Листинг 4.6 (*продолжение*)

```
                                    pisl->Parameters.DeviceIoControl
                                                    .InputBufferLength,

                                    pIrp->AssociatedIrp.SystemBuffer,
                                    pisl->Parameters.DeviceIoControl
                                                    OutputBufferLength,
                                    &dInfo);
            break;
            }
        }
    pIrp->IoStatus.Status       = ns;
    pIrp->IoStatus.Information  = dInfo;

    IoCompleteRequest (pIrp, IO_NO_INCREMENT);
    return ns;
    }
```

Обработчик IOCTL в листинге 4.6 весьма прост — он вызывает функцию SpyDispatcher() с параметрами, извлекаемыми из структуры IRP и текущего стека ввода-вывода, указатель на который встроен в IRP. Функция SpyDispatcher(), показанная в листинге 4.7, ожидает следующие аргументы:

- pDeviceContext — контекст устройства драйвера. Предоставленная мастером драйверов базовая структура Device_Context содержит только указатели на объекты устройства и драйвера (см. листинг 3.4). Драйвер слежения добавляет в нее еще несколько членов для своей работы;
- dCode содержит код IOCTL, передающий команду, которую должно выполнить устройство просмотра. Код IOCTL представляет собой 32-битное целое, состоящее из четырех битовых полей, как показано на рис. 4.6;
- pInput указывает на буфер входных данных IOCTL;
- dInput — размер входного буфера;
- pOutput указывает на буфер выходных данных IOCTL;
- dOutput — размер выходного буфера;
- pdInfo указывает на переменную типа DWORD, через которую должно быть передано число байт, записанных в выходной буфер.

В зависимости от используемого метода IOCTL входной и выходной буферы передаются системой драйверу по-разному. Ввод-вывод драйвера слежения буферизован, система должна скопировать входные данные в безопасный буфер, автоматически выделяемый системой, а при возврате — скопировать указанное количество данных из того же системного буфера в выходной буфер вызывающей программы. Важно иметь в виду, что в этом случае входной и выходной буферы перекрываются, поэтому перед записью в буфер обработчик IOCTL должен сохранить все входные данные, которые ему могут понадобиться в дальнейшем. Указатель на этот буфер ввода-вывода хранится в члене SystemBuffer объединения AssociatedIrp внутри структуры IRP (см. ntddk.h). Размеры входного и выходного буферов хранятся в совершенно различных участках IRP — они входят в член DeviceIoControl объединения Parameters внутри области памяти текущего стека IRP

и называются соответственно InputBufferLength и OutputBufferLength. Вложенная структура DeviceIoControl также содержит код IOCTL в своем члене IoControlCode. Дополнительная информация о методах IOCTL Windows NT/2000 и том, как они передают данные, приведена в моей статье «A Spy Filter Driver for Windows NT» в *Windows Developer's Journal* (Schreiber, 1997).

Листинг 4.7. Внутренний диспетчер команд драйвера слежения

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT    pDeviceContext,
                        DWORD              dCode,
                        PVOID              pInput,
                        DWORD              dInput,
                        PVOID              pOutput,
                        DWORD              dOutput,
                        PDWORD             pdInfo)
    {
    SPY_MEMORY_BLOCK        smb;
    SPY_PAGE_ENTRY spe;
    SPY_CALL_INPUT sci;
    PHYSICAL_ADDRESS        pa;
    DWORD                   dValue, dCount;
    BOOL                    fReset, fPause, fFilter, fLine;
    PVOID                   pAddress;
    PBYTE                   pbName;
    HANDLE                  hObject;
    NTSTATUS                ns = STATUS_INVALID_PARAMETER;

    MUTEX_WAIT  (pDeviceContext->kmDispatch);

    *pdInfo = 0;

    switch (dCode)
        {
        case SPY_IO_VERSION_INFO:
            {
            ns = SpyOutputVersionInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_ID_OS_INFO:
            {
            ns = SpyOutputOsInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_SEGMENT:
            {
            if ((ns = SpyInputDword (&dValue,
                                     pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputSegment (dValue,
                                       pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_INTERRUPT:
            {
```

продолжение

Листинг 4.7 (*продолжение*)

```
            if ((ns = SpyInputDword (&dValue,
                                     pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputInterrupt (dValue,
                                         pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PHYSICAL:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                pa = MmGetPhysicalAddress (pAddress);

                ns = SpyOutputBinary (&pa, PHYSICAL_ADDRESS_,
                                      pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_CPU_INFO:
            {
            ns = SpyOutputCpuInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_PDE_ARRAY:
            {
            ns = SpyOutputBinary (X86_PDE_ARRAY, SPY_PDE_ARRAY_,
                                  pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_PAGE_ENTRY:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                SpyMemoryPageEntry (pAddress, &spe);

                ns = SpyOutputBinary (&spe, SPY_PAGE_ENTRY_,
                                      pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_MEMORY_DATA:
            {
            if ((ns = SpyInputMemory (&smb,
                                      pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputMemory (&smb,
                                      pOutput, dOutput, pdInfo);
                }
```

```
        break;
        }
    case SPY_IO_MEMORY_BLOCK:
        {
        if ((ns = SpyInputMemory (&smb,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputBlock (&smb,
                                 pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HANDLE_INFO:
        {
        if ((ns = SpyInputHandle (&hObject,
                                  pInput, dInput))
            == STATUS_SUCCESS)
            {
            ns = SpyOutputHandleInfo (hObject,
                                      pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_INFO:
        {
        ns = SpyOutputHookInfo (pOutput, dOutput,
                                pdInfo);
        break;
        }
    case SPY_IO_HOOK_INSTALL:
        {
        if (((ns = SpyInputBool (&fReset,
                                 pInput, dInput))
             == STATUS_SUCCESS)
            &&
            ((ns = SpyHookInstall (fReset, &dCount))
             == STATUS_SUCCESS))
            {
            ns = SpyOutputDword (dCount,
                                 pOutput, dOutput, pdInfo);
            }
        break;
        }
    case SPY_IO_HOOK_REMOVE:
        {
        if (((ns = SpyInputBool (&fReset,
                                 pInput, dInput))
             == STATUS_SUCCESS)
            &&
            ((ns = SpyHookRemove (fReset, &dCount))
             == STATUS_SUCCESS))
            {
            ns = SpyOutputDword (dCount,
                                 pOutput, dOutput, pdInfo);
            }
```

продолжение ⇩

Листинг 4.7 (*продолжение*)

```
            break;
            }
        case SPY_IO_HOOK_PAUSE:
            {
            if ((ns = SpyInputBool (&fPause,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                fPause = SpyHookPause (fPause);

                ns = SpyOutputBool (fPause,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_FILTER:
            {
            if ((ns = SpyInputBool (&fFilter,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                fFilter = SpyHookFilter (fFilter);

                ns = SpyOutputBool (fFilter,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_RESET:
            {
            SpyHookReset ();
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_HOOK_READ:
            {
            if ((ns = SpyInputBool (&fLine,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputHookRead (fLine,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_WRITE:
            {
            SpyHookWrite (pInput, dInput);
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_MODULE_INFO:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
```

```
                == STATUS_SUCCESS)
                {
                ns = SpyOutputModuleInfo (pbName,
                                          pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_HEADER:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeHeader (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_EXPORT:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeExport (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_SYMBOL:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeSymbol (pbName,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_CALL:
            {
            if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_,
                                      pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputCall (&sci,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
    }
MUTEX_RELEASE  (pDeviceContext->kmDispatch);
return ns;
}
```

Основной заголовочный файл комплекта DDK ntddk.h, так же как и файл Win32 winioctl.h в комплекте Platform SDK, определяет простой, но чрезвычайно удобный макрос CTL_CODE(), показанный в листинге 4.8, нужный для генерирования кодов IOCTL в соответствии с диаграммой на рис. 4.6. Четыре части макроса служат следующим целям:

1. DeviceType — это 16-битный идентификатор типа устройства. В файле ntddk.h перечислен ряд типов устройств, в символьном виде представленных константами FILE_DEVICE_*. Microsoft резервирует диапазон с 0x0000 по 0x7FFF для внутреннего использования, а диапазон 0x8000–0xFFFF доступен для разработчиков. Драйвер слежения определяет свой собственный идентификатор устройства FILE_DEVICE_SPY и устанавливает его в значение 0x8000.
2. Access задает двухбитовое значение для проверки возможности доступа, задающее требуемые права доступа для работы IOCTL. Возможны следующие значения: FILE_ANY_ACCESS (0), FILE_READ_ACCESS (1), FILE_WRITE_ACCESS (2) и комбинация двух последних: FILE_READ_ACCESS | FILE_WRITE_ACCESS (3). За более подробной информацией обратитесь к файлу ntddk.h.
3. Function — это 12-битный идентификатор, задающий операцию, которую должно выполнить устройство. Microsoft резервирует значения с 0x000 по 0x7FF для внутреннего использования и оставляет для разработчиков диапазон 0x800–0xFFF. Идентификаторы функций IOCTL, распознаваемые устройством просмотра, берутся из последнего пула номеров.
4. Method состоит из двух бит, определяя один из четырех возможных методов передачи ввода-вывода с именами METHOD_BUFFERED (0), METHOD_IN_DIRECT (1), METHOD_OUT_DIRECT (2) и METHOD_NEITHER (3), расположенных в файле ntddk.h. Драйвер слежения для всех запросов использует способ METHOD_BUFFERED, обладающий высокой безопасностью, но и весьма медлительный, поскольку данные копируются из системного буфера в клиентский и обратно. Поскольку операции ввода-вывода драйвера слежения не критичны по времени, выбор безопасного метода — неплохое решение. Если вас интересует дополнительная информация о других методах, обратитесь к моей статье о программе просмотра памяти с фильтрацией.

Листинг 4.8. Макроопределение CTL_CODE(), генерирующее управляющие коды ввода-вывода

```
#define CTL_CODE(DeviceType, Function, Method, Access) \
                (((DeviceType) << 16) | ((Access) << 14) | \
                ((Function) << 2) | (Method))
```

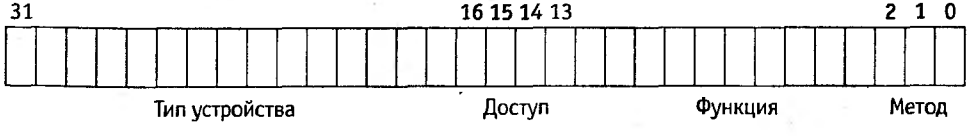

Рис. 4.6. Схема управляющего кода устройства ввода-вывода

В табл. 4.2 сведена информация обо всех функциях IOCTL, поддерживаемых модулем w2k_spy.sys. Функции с идентификаторами с 0 по 10 определяют элементарные операции для исследования памяти, которых достаточно для выполнения широкого круга задач, они будут обсуждены позже в этой главе. Оставшиеся функции с идентификаторами начиная с 11 и выше принадлежат к различным группам IOCTL, которые будут подробно описаны в следующих главах, посвященных перехватчикам (hook) вызовов Native API и ядра из пользовательского режима. Обратите внимание на то, что некоторым кодам IOCTL требуется доступ на запись, что отмечается установкой бита №15 (рис. 4.6). Таким образом, все команды IOCTL с кодами вида 0x80006nnn могут выполняться посредством обработчика с доступом только на чтение, а код 0x8000Ennn требует обработчика с доступом на чтение и на запись. Права доступа, как правило, запрашиваются при вызове CreateFile(), который открывает устройство, задавая комбинацию флагов GENERIC_READ и GENERIC_WRITE для аргумента dwDesiredAccess.

Имена функций в самом левом столбце табл. 4.2 также встречаются как варианты выбора большого оператора switch/case в функции SpyDispatcher() в листинге 4.7. Эта функция сначала получает мьютекс (mutex) диспетчера устройства, чтобы гарантировать выполнение только одного запроса за раз, если к устройству обратятся несколько клиентов или многопоточное приложение. MUTEX_WAIT() — это макрос-оболочка для KeWaitForMutexObject(), он принимает не менее пяти аргументов. KeWaitForMutexObject() — это тоже макрос, передающий свои аргументы KeWaitForSingleObject(). В листинге 4.9 показан макрос MUTEX_WAIT() и родственные ему макроопределения MUTEX_RELEASE() и MUTEX_INITIALIZE(). После захвата объекта мьютекса функция SpyDispatcher() переключается на один из коротких участков кода в соответствии с полученным кодом IOCTL. После окончания работы функция освобождает мьютекс и возвращает код состояния вызывающей функции.

Функция SpyDispatcher() использует ряд вспомогательных функций для чтения входных параметров, получения запрашиваемых данных и записи данных в выходной буфер вызывающей функции. Как уже упоминалось, драйвер режима ядра должен чрезвычайно внимательно относиться ко всем получаемым параметрам пользовательского режима. С точки зрения драйвера весь код пользовательского режима ошибочен и не имеет других намерений, кроме как вывести систему из строя. Такая перестраховка имеет смысл — достаточно малейшей описки, чтобы вся система немедленно остановилась и появился синий экран смерти (BSOD). Поэтому если клиентское приложение утверждает: «Вот мой буфер, он может занимать не более 4096 байт», драйвер этому не верит — ни тому, что буфер указывает на допустимую область памяти, ни тому, что размер буфера правильный. В случае IOCTL с буферизованным вводом-выводом (то есть если часть Method кода IOCTL равна METHOD_BUFFERED), система берет на себя заботу о проверке правильности и выделяет буфер, достаточно большой для того, чтобы в нем поместились входные и выходные данные. Тем не менее при остальных способах передачи, особенно при METHOD_NEITHER, когда уже заданные указатели на буфер передаются драйверу из пользовательского режима, необходима большая предусмотрительность.

Таблица 4.2. Функции IOCTL, реализованные в драйвере слежения

Имя функции	ID	Код IOCTL	Описание
SPY_IO_VERSION_INFO	0	0x80006000	Возвращает информацию о версии программы просмотра памяти
SPY_IO_OS_INFO	1	0x80006004	Возвращает информацию об операционной системе
SPY_IO_SEGMENT	2	0x80006008	Возвращает свойства сегмента
SPY_IO_INTERRUPT	3	0x8000600C	Возвращает свойства шлюза прерываний
SPY_IO_PHYSICAL	4	0x80006010	Преобразование адреса из линейного в физический
SPY_IO_CPU_INFO	5	0x80006014	Возвращает значения специальных регистров процессора
SPY_IO_PDE_ARRAY	6	0x80006018	Возвращает массив PDE по адресу 0xC0300000
SPY_IO_PAGE_ENTRY	7	0x8000601C	Возвращает PDE или PTE линейного адреса
SPY_IO_MEMORY_DATA	8	0x80006020	Возвращает содержимое блока памяти
SPY_IO_MEMORY_BLOCK	9	0x80006024	Возвращает содержимое блока памяти
SPY_IO_HANDLE_INFO	10	0x80006028	Получает свойства объекта из дескриптора объекта
SPY_IO_HOOK_INFO	11	0x8000602C	Возвращает информацию о перехватчиках Native API
SPY_IO_HOOK_INSTALL	12	0x8000E030	Устанавливает перехватчики API
SPY_IO_HOOK_REMOVE	13	0x8000E034	Удаляет перехватчики API
SPY_IO_HOOK_PAUSE	14	0x8000E038	Приостанавливает/возобновляет протоколирование перехвата
SPY_IO_HOOK_FILTER	15	0x8000E03C	Включает/отключает фильтр журнала перехвата
SPY_IO_HOOK_RESET	16	0x8000E040	Очищает журнал перехвата
SPY_IO_HOOK_READ	17	0x80006044	Читает данные из журнала перехвата
SPY_IO_HOOK_WRITE	18	0x8000E048	Записывает данные в журнал перехвата
SPY_IO_MODULE_INFO	19	0x8000604C	Возвращает информацию о загруженных системных модулях
SPY_IO_PE_HEADER	20	0x80006050	Возвращает данные из IMAGE_NT_HEADERS
SPY_IO_PE_EXPORT	21	0x80006054	Возвращает данные из IMAGE_EXPORT_DIRECTORY
SPY_IO_PE_SYMBOL	22	0x80006058	Возвращает адрес экспортируемого системного идентификатора
SPY_IO_CALL	23	0x8000E05C	Вызывает функцию из загруженного модуля

Листинг 4.9. Макрос управлением объектом ядра мьютекс

```
#define MUTEX_INITIALIZE(_mutex) \
          KeInitializeMutex \
               (&(_mutex), 0)

#define MUTEX_WAIT(_mutex) \
          KeWaitForMutexObject \
               (&(_mutex), Executive, KernelMode, FALSE, NULL)

#define MUTEX_RELEASE(_mutex) \
          KeReleaseMutex \
               (&(_mutex), FALSE)
```

Хотя устройство просмотра применяет буферизованный ввод-вывод, ему нужно проверять допустимость входных и выходных параметров. Может оказаться, что клиентское приложение передает меньше данных, чем требуется, или предостав-

ляет выходной буфер, недостаточно большой для выходных данных. Система не может отследить эти семантические проблемы, поскольку она не знает, какой вид данных передается в сообщении IOCTL. Поэтому для обмена данными с буферами ввода-вывода SpyDispatcher() вызывает вспомогательные функции SpyInput*() и SpyOutput, которые выполняют запрашиваемую операцию, только если размер буфера удовлетворяет требованиям для этой операции. Листинг 4.10 демонстрирует основные функции ввода, а листинг 4.11 — функции вывода. Львиную долю работы выполняют функции SpyInputBinary() и SpyOutputBinary(): они проверяют размер буфера, и если все в порядке, копируют запрашиваемое количество данных при помощи функции библиотеки времени выполнения Windows 2000 RtlCopyMemory(). Остальные функции представляют собой простые оболочки для распространенных типов данных DWORD, BOOL, PVOID и HANDLE. Функция SpyOuBlock(), помимо этого, копирует блок данных, указанный вызывающей функцией, в структуре SPY_MEMORY_BLOCK, после проверки того, что все байты в указанном диапазоне читаются. Функции SpyInput*() возвращают STATUS_INVALID_BUFFER_SIZE при передаче неполных входных данных, а функции SpyOutput*() возвращают STATUS_BUFFER_TOO_SMALL, если выходной буфер меньше, чем необходимо.

Листинг 4.10. Чтение входных данных из буфера IOCTL

```
NTSTATUS SpyInputBinary (PVOID  pData,
                         DWORD  dData,
                         PVOID  pInput,
                         DWORD  dInput)
      {
      NTSTATUS ns = STATUS_INVALID_BUFFER_SIZE;

      if (dData <= dInput)
          {
          RtlCopyMemory (pData, pInput, dData);
          ns = STATUS_SUCCESS;
          }
      return ns;
      }

//  ------------------------------------------------------

NTSTATUS SpyInputDword (PDWORD  pdValue,
                        PVOID   pInput,
                        DWORD   dInput)
      {
      return SpyInputBinary (pdValue, DWORD_, pInput, dInput);
      }

//  ------------------------------------------------------

NTSTATUS SpyInputBool (PBOOL  pfValue,
                       PVOID  pInput,
                       DWORD  dInput)
      {
      return SpyInputBinary (pfValue, BOOL_, pInput, dInput);
      }
```

продолжение

Листинг 4.10 (*продолжение*)

```
//  ---------------------------------------------------

NTSTATUS SpyInputPointer (PPVOID  ppAddress,
                          PVOID   pInput,
                          DWORD   dInput)
      {
      return SpyInputBinary (ppAddress, PVOID_, pInput, dInput);
      }

//  ---------------------------------------------------

NTSTATUS SpyInputHandle (PHANDLE  phObject,
                         PVOID    pInput,
                         DWORD    dInput)
      {
      return SpyInputBinary (phObject, HANDLE_, pInput, dInput);
      }
```

Листинг 4.11. Запись выходных данных в буфер IOCTL

```
NTSTATUS SpyOutputBinary (PVOID   pData,
                          DWORD   dData,
                          PVOID   pOutput,
                          DWORD   dOutput,
                          PDWORD  pdInfo)
      {
      NTSTATUS ns = STATUS_BUFFER_TOO_SMALL;

      *pdInfo = 0;

      if (dData <= dOutput)
          {
          RtlCopyMemory (pOutput, pData, *pdInfo = dData);
          ns = STATUS_SUCCESS;
          }
      return ns;
      }

//  ---------------------------------------------------

NTSTATUS SpyOutputBlock (PSPY_MEMORY_BLOCK     psmb,
                         PVOID                 pOutput,
                         DWORD                 dOutput,
                         PDWORD                pdInfo)
      {
      NTSTATUS ns = STATUS_INVALID_PARAMETER;

      if (SpyMemoryTestBlock (psmb->pAddress, psmb->dBytes))
          {
          ns = SpyOutputBinary (psmb->pAddress, psmb->dBytes,
                                pOutput, dOutput, pdInfo);
          }
      return ns;
      }

//  ---------------------------------------------------
```

```
NTSTATUS SpyOutputDword (DWORD   dValue.
                         PVOID   pOutput.
                         DWORD   dOutput.
                         PDWORD  pdInfo)
     {
     return SpyOutputBinary (&dValue, DWORD_.
                             pOutput. dOutput, pdInfo);
     }

//  ------------------------------------------------------

NTSTATUS SpyOutputBool (BOOL     fValue.
                        PVOID   pOutput.
                        DWORD   dOutput.
                        PDWORD  pdInfo)
     {
     return SpyOutputBinary (&fValue. BOOL_.
                                     pOutput. dOutput, pdInfo);
     }

//  ------------------------------------------------------

NTSTATUS SpyOutputPointer (PVOID   pValue.
                                     PVOID    pOutput.
                                     DWORD    dOutput.
                                     PDWORD   pdInfo)
     {
     return SpyOutputBinary (&pValue, PVDID_.
                                     pOutput. dOutput, pdInfo);
     }
```

Вы могли заметить, что функция SpyDispatcher() из листинга 4.7 обращается несколько к большему числу функций SpyInput*() и SpyOutput*(). Хотя эти функции в конечном итоге основаны на SpyInputBinary() и SpyOutputBinary(), они несколько сложнее основных функций из листингов 4.10 и 4.11 и поэтому будут рассмотрены отдельно чуть позже (в этой главе). Итак, давайте начнем с самого начала функции SpyDispatcher() и пройдем по ветвям оператора switch/case шаг за шагом.

Функция IOCTL SPY_IO_VERSION_INFO

Функция IOCTL SPY_IO_VERSION заполняет возвращаемую вызывающей функцией структуру SPY_VERSION_INFO данными о самом драйвере слежения. Функция не требует входных параметров и пользуется вспомогательной функцией SpyOutputVersionInfo(), довольно элементарной функцией, определение которой приведено в листинге 4.12 вместе с определением структуры SPY_VERSION_INFO. Она задает значение для члена структуры dVersion, равное значению определенной в файле w2k_spy.h константы SPY_VERSION (на данный момент оно равно 100, означая V1.00), и копирует в член структуры awName имя драйвера, хранимое в символьной константе DRV_NAME («SBS Windows 2000 Spy Device»). Основной номер версии получается путем деления dVersion на 100. Остаток будет равен номеру подверсии.

Листинг 4.12. Получение информации о версии драйвера слежения

```
typedef struct _SPY_VERSION_INFO
      {
      DWORD dVersion;
      WORD  awName [SPY_NAME];
      }
      SPY_VERSION_INFO,  *PSPY_VERSION_INFO,
                         **PPSPY_VERSION_INFO;
#define SPY_VERSION_INFO_ sizeof (SPY_VERSION_INFO)

//  ------------------------------------------------------

NTSTATUS SpyOutputVersionInfo (PVOID    pOutput,
                               DWORD    dOutput,
                               PDWORD   pdInfo)

      {
      SPY_VERSION_INFO svi;

      svi.dVersion = SPY_VERSION;

      wcscpyn (svi.awName, USTRING (CSTRING (DRV_NAME)),
               SPY_NAME);

      return SpyOutputBinary (&svi, SPY_VERSION_INFO_,
                              pOutput, dOutput, pdInfo);
      }
```

Функция IOCTL SPY_IO_OS_INFO

Функция IOCTL SPY_IO_OS_INFO гораздо интереснее предыдущей. Это тоже функция только с выходными данными, не ожидающая никаких входных значений и заполняющая возвращаемую вызывающей функцией структуру SPY_OS_INFO значениями ряда внутренних параметров операционной системы. В листинге 4.13 показано определение этой структуры и вспомогательной функции SpyOutputOsInfo(), вызываемой диспетчером. Некоторые члены структуры просто устанавливаются в константы, позаимствованные из заголовочных файлов DDK и файла w2k_spy.h, другие получают «реальные» значения, прочитанные из нескольких внутренних переменных и структур ядра. В главе 2 вы познакомились с переменными NtBuildNumber и NtGlobalFlag, экспортируемыми модулем ntoskrnl.exe (см. табл. Б.1 в приложении Б). В отличие от других экспортируемых символьных идентификаторов Nt* они указывают не на функции API, а на переменные в секции .data ядра. В мире Win32 экспорт переменных весьма необычен, но, несмотря на это, некоторые модули ядра Windows 2000 применяют эту технику. Модуль ntoskrnl.exe экспортирует не менее чем 55 переменных, ntdll.dll предоставляет четыре, а hal.dll — одну. Функция SpyOutputOsInfo() копирует из набора переменных модуля ntoskrnl.exe в выходной буфер переменные MmHighestUserAddress, MmUserProbeAddress, MmSystemRangeStart, NtGlobalFlag, KeI386MachineType, KeNumberProcessors и NtBuildNumber.

При импорте данных из одного модуля в другой необходимо соответствующим образом проинформировать компилятор и компоновщик при помощи ключевого слова extern. Компоновщик при этом создаст точку входа в секции импорта моду-

ля, вместо того чтобы пытаться преобразовать идентификатор в конкретный адрес. Некоторые объявления extern уже включены в файл ntddk.h, пропущенные включены в листинг 4.13.

Листинг 4.13. Получение информации об операционной системе

```
typedef struct _SPY_DS_INFO
      {
      DWORD          dPageSize;
      DWORD          dPageShift;
      DWORD          dPtiShift;
      DWORD          dPdiShift;
      DWORD          dPageMask;
      DWORD          dPtiMask;
      DWORD          dPdiMask;
      PX86_PE        PteArray;
      PX86_PE        PdeArray;
      PVOID          pLowestUserAddress;
      PVOID          pThreadEnvironmentBlock;
      PVOID          pHighestUserAddress;
      PVOID          pUserProbeAddress;
      PVOID          pSystemRangeStart;
      PVOID          pLowestSystemAddress;
      PVOID          pSharedUserData;
      PVOID          pProcessorControlRegion;
      PVOID          pProcessorControlBlock;
      DWORD          dGlobalFlag;
      DWORD          dI386MachineType;
      DWORD          dNumberProcessors;
      DWORD          dProductType;
      DWORD          dBuildNumber;
      DWORD          dNtMajorVersion;
      DWORD          dNtMinorVersion;
      WORD           awNtSystemRoot [MAX_PATH];
      }
      SPY_OS_INFO, *PSPY_OS_INFO, **PPSPY_OS_INFO;

#define SPY_DS_INFO_ sizeof (SPY_OS_INFO)

//  ------------------------------------------------------

extern PWORD    NlsAnsiCodePage;
extern PWORD    NlsOemCodePage;
extern PWORD    NtBuildNumber;
extern PDWORD   NtGlobalFlag;
extern PDWORD   KeI386MacnineType;

//  ------------------------------------------------------
NTSTATUS SpyOutputDsInfo (PVOID    pOutput,
                          DWORD    dOutput,
                          PDWORD   pdInfo)
      {
      SPY_SEGMENT        ss;
      SPY_OS_INFO        soi;
      NT_PRODUCT_TYPE    NtProductType;
      PKPCR              pkpcr;
```

продолжение

Листинг 4.13 (*продолжение*)

```
    NtProductType = (SharedUserData->ProductTypeIsValid
                        ? SharedUserData->NtProductType
                        : 0);

    SpySegment (X86_SEGMENT_FS, 0, &ss);
    pkpcr = ss.pBase;

    soi.dPageSize                   = PAGE_SIZE;
    soi.dPageShift                  = PAGE_SHIFT;
    soi.dPtiShift                   = PTI_SHIFT;
    soi.dPdiShift                   = PDI_SHIFT;
    soi.dPageMask                   = X86_PAGE_MASK;
    soi.dPtiMask                    = X86_PTI_MASK;
    soi.dPdiMask                    = X86_PDI_MASK;
    soi.PteArray                    = X86_PTE_ARRAY;
    soi.PdeArray                    = X86_PDE_ARRAY;
    soi.pLowestUserAddress          = MM_LOWEST_USER_ADDRESS;
    soi.pThreadEnvironmentBlock     = pkpcr->NtTib.Self;
    soi.pHighestUserAddress         = *MmHighestUserAddress;
    soi.pUserProbeAddress           = (PVOID) *MmUserProbeAddress;
    soi.pSystemRangeStart           = *MmSystemRangeStart;
    soi.pLowestSystemAddress        = MM_LOWEST_SYSTEM_ADDRESS;
    soi.pSharedUserData             = SharedUserData;
    soi.pProcessorControlRegion     = pkpcr;
    soi.pProcessorControlBlock      = pkpcr->Prcb;
    soi.dGlobalFlag                 = *NtGlobalFlag;
    soi.dI386MachineType            = *KeI386MachineType;
    soi.dNumberProcessors           = *KeNumberProcessors;
    soi.dProductType                = NtProductType;
    soi.dBuildNumber                = *NtBuildNumber;
    soi.dNtMajorVersion             = SharedUserData->
                                            NtMajorVersion;
    soi.dNtMinorVersion             = SharedUserData->
                                            NtMinorVersion;

    wcscpyn (soi.awNtSystemRoot, SharedUserData->NtSystemRoot,
            MAX_PATH);

    return SpyOutputBinary (&soi, SPY_OS_INFO_,
                            pOutput, dOutput, pdInfo);
    }
```

Оставшиеся члены структуры SPY_OS_INFO заполняются значениями из системных структур данных, расположенных в памяти. Например, SpyOutputOsInfo() присваивает базовый адрес расположенной в ядре управляющей области процессора (Kernel's Processor Control Region, KCPR) переменной-члену pProcessorControlRegion. Это очень важная структура данных, содержащая множество часто используемых данных отдельных потоков, поэтому она расположена в своем собственном сегменте памяти, на который указывает регистр процессора PS. И Windows NT 4.0, и Windows 2000 устанавливают PS в значение, указывающее на линейный адрес 0xFFDFF000 в режиме ядра. Для запроса базового адреса сегмента PS в линейном адресном пространстве функция SpyOutputOsInfo() вызывает функцию SpySegment(), которая будет обсуждена позже. В этом сегменте также находится расположенный в ядре

управляющий блок процессора (Kernel's Processor Control Block, KPRCB), на который указывает член Prcb структуры KPCR, сразу за которым расположена структура CONTEXT, содержащая низкоуровневую информацию состояния процессора для текущего потока. Определения структур KPCR, KPRCB и CONTEXT можно найти в заголовочном файле ntddk.h. В последующих частях главы приведен более подробный материал по этой теме.

В листинге 4.14 упомянута еще одна внутренняя структура данных — SharedUserData. В действительности это не что иное, как «хорошо известный адрес», приведенный к типу «указатель на структуру» (см. листинг 4.14). Хорошо известные адреса — это адреса в пределах линейного адресного пространства, которые устанавливаются на этапе компиляции и поэтому не меняются со временем или с изменением конфигурации. Ясно, что SharedUserData — это указатель на структуру KUSER_SHARED_DATA, расположенную по фиксированному линейному адресу 0xFFDF0000. Эта область памяти используется совместно приложениями пользовательского режима и системой и содержит такие интересные данные, как номер версии операционной системы, который SpyOutputOsInfo() копирует в члены dNtMajorVersion и dNtMinorVersion структуры SPY_OS_INFO вызывающей функции. Как я покажу далее, структура KUSER_ SHARED_DATA зеркально отображается по адресу 0x7FFE0000, где она доступна коду режима пользователя.

После объяснения функций IOCTL драйвера слежения будет описано демонстрационное приложение, отображающее возвращаемые данные на экране.

Листинг 4.14. Определение SharedUserData

```
#define KI_USER_SHARED_DATA 0xffdf0000
#define SharedUserData ((KUSER_SHARED_DATA * const) \
                                    KI_USER_SHARED_DATA)
```

Функция IOCTL SPY_IO_SEGMENT

С данного момента обсуждение становится по-настоящему интересным. Функция SPY_IO_SEGMENT по заданному селектору запрашивает свойства сегмента, осуществляя несколько операций очень низкого уровня. SpyDispatcher() сначала вызывает SpyInputDword(), чтобы получить значение селектора, переданного вызывающим приложением. Как вы, возможно, помните, селекторы — это 16-разрядные величины. Однако я стараюсь по мере возможности избегать 16-разрядных типов данных, поскольку «родным» размером слова процессоров i386 в 32-разрядном режиме является 32-разрядный тип DWORD. Поэтому я расширил значение селектора до DWORD, в котором старшие 16 бит всегда равны нулю. Если SpyInputDword() сообщает об успешном выполнении, вызывается функция SpyOutputSegment(), показанная в листинге 4.15. Эта функция просто возвращает то значение, которое возвращает вспомогательная функция SpySegment(), включенная в листинг 4.15. По сути, SpySegment() заполняет структуру SPY_SEGMENT, определенную в начале листинга 4.15. Эта структура состоит из значений селектора и дескриптора в виде структуры X86_SELECTOR и 64-битного объединения X86_DESCRIPTOR (листинг 4.2), линейного базового адреса соответствующего сегмента, границы сегмента (то есть размер

сегмента минус один) и флага fOk, показывающего допустимость данных в структуре SPY_SEGMENT. Последний флаг требуется в контексте других функций (например, SPY_IO_CPU_INFO), которые возвращают свойства нескольких сегментов сразу. В этом случае флаг fOk дает возможность вызывающей функции отфильтровать все неверные сегменты в выходных данных.

Листинг 4.15. Запрос свойств сегмента

```
typedef struct _SPY_SEGMENT
      {
      X86_SELECTOR        Selector;
      X86_DESCRIPTOR      Descriptor;
      PVOID               pBase;
      DWORD               dLimit;
      BOOL                fOk;
      }
      SPY_SEGMENT, *PSPY_SEGMENT, **PPSPY_SEGMENT;

#define SPY_SEGMENT_ sizeof (SPY_SEGMENT)

//  -------------------------------------------------------
NTSTATUS SpyOutputSegment (DWORD   dSelector,
                           PVOID   pOutput,
                           DWORD   dOutput,
                           PDWORD  pdInfo)
      {
      SPY_SEGMENT ss;

      SpySegment (X86_SEGMENT_OTHER, dSelector, &ss);

      return SpyOutputBinary (&ss, SPY_SEGMENT_,
                              pOutput, dOutput, pdInfo);
      }

//  -------------------------------------------------------

BOOL SpySegment (DWORD          dSegment,
                 DWORD          dSelector,
                 PSPY_SEGMENT   pSegment)
      {
      BOOL fOk = FALSE;

      if (pSegment != NULL)
           {
           fOk = TRUE;

           if (!SpySelector   (dSegment, dSelector,
                               &pSegment->Selector))
                {
                fOk = FALSE;
                }
           if (!SpyDescriptor (&pSegment->Selector,
                               &pSegment->Descriptor))
                {
                fOk = FALSE;
                }
```

```
        pSegment->pBase  =
             SpyDescriptorBase   (&pSegment->Descriptor);

        pSegment->dLimit =
             SpyDescriptorLimit  (&pSegment->Descriptor);

        pSegment->fOk = fOk;
        }
    return fOk;
    }
```

SpySegment() использует несколько вспомогательных функций, получающих значения частей итоговой структуры SPY_SEGMENT. Сначала SpySelector() копирует значение селектора в структуру X86_SELECTOR, переданную в качестве параметра (листинг 4.16). Если первый аргумент dSegment установлен в значение X86_SEGMENT_OTHER, считается, что аргумент dSelector указывает допустимое значение селектора, поэтому это значение просто назначается члену wValue возвращаемой структуры. В ином случае dSelector игнорируется и dSegment используется в операторе switch/case для выбора одного из сегментных регистров процессора или его регистра задачи TR. Обратите внимание на необходимость включения небольших ассемблерных вставок — в языке C нет стандартных средств для доступа к зависящим от процессора ресурсам, таким как сегментные регистры.

Листинг 4.16. Получение значений селектора

```
#define  X86_SEGMENT_OTHER          0
#define  X86_SEGMENT_CS             1
#define  X86_SEGMENT_DS             2
#define  X86_SEGMENT_ES             3
#define  X86_SEGMENT_FS             4
#define  X86_SEGMENT_GS             5
#define  X86_SEGMENT_SS             6
#define  X86_SEGMENT_TSS            7

//  ------------------------------------------------------

BOOL SpySelector (DWORD            dSegment,
                  DWORD            dSelector,
                  PX86_SELECTOR    pSelector)
    {
    X86_SELECTOR Selector = {0, 0};
    BOOL           fOk         = FALSE;

    if (pSelector != NULL)
        {
        fOk = TRUE;

        switch (dSegment)
            {
            case  X86_SEGMENT_OTHER:
                {
                if (fOk = ((dSelector >> X86_SELECTOR_SHIFT)
                             <= X86_SELECTOR_LIMIT))
                    {
```

продолжение ⇩

Листинг 4.16 (*продолжение*)

```
                    Selector.wValue = (WORD) dSelector;
                    }
                break;
                }
            case X86_SEGMENT_CS:
                {
                __asm mov Selector.wValue, cs
                break;
                }
            case X86_SEGMENT_DS:
                {
                __asm mov Selector.wValue, ds
                break;
                }
            case X86_SEGMENT_ES:
                {
                __asm mov Selector.wValue, es
                break;
                }
            case X86_SEGMENT_FS:
                {
                __asm mov Selector.wValue, fs
                break;
                }
            case X86_SEGMENT_GS:
                {
                __asm mov Selector.wValue, gs
                break;
                }
            case X86_SEGMENT_SS:
                {
                __asm mov Selector.wValue, ss
                break;
                }
            case X86_SEGMENT_TSS:
                {
                __asm str Selector.wValue
                break;
                }
            default:
                {
                fOk = FALSE;
                break;
                }
            }
        RtlCopyMemory (pSelector, &Selector,
                       X86_SELECTOR_);
        }
    return fOk;
    }
```

Функция SpyDescriptor() считывает 64-битный дескриптор, на который указывает селектор сегмента (листинг 4.17). Как вы, возможно, помните, все селекторы содержат бит индикатора таблицы (TI, Table Indicator), определяющий, ссылается ли селектор на дескриптор в глобальной таблице дескрипторов (Global Descriptor

Table, GDT, TI=0) или локальной таблице дескрипторов (Local Descriptor Table, LDT, TI=1). Первая половина листинга 4.17 относится к случаю таблицы LDT. Сначала при помощи ассемблерных инструкций SLDT и SGDT считываются соответственно значение селектора LDT, граница сегмента и базовый адрес GDT. Помните, что линейный базовый адрес таблицы GDT задается явно, в то время как к таблице LDT необходимо обращаться через селектор, указывающий на запись в таблице GDT. Поэтому SpyDescriptor() сначала проверяет значение селектора LDT. Если он не соответствует селектору нулевого сегмента и не указывает за границу таблицы GDT, то для получения основных свойств LDT вызываются функции SpyDescriptorType(), SpyDescriptorLimit() и SpyDescriptorBase(), расположенные внизу листинга 4.17:

- SpyDescriptorType() возвращает значения полей дескриптора Type и S (см. листинг 4.2). Селектор LDT должен указывать на системный дескриптор с типом данных X86_DESCRIPTOR_SYS_LDT(2);
- SpyDescriptorLimit() собирает границу сегмента из битовых полей дескриптора Limit1 и Limit2. Если флаг дескриптора *G* показывает масштаб в 4 Кбайт, значение сдвигается влево на 12 бит, а оставшиеся справа разряды заполняются единицами;
- SpyDescriptorBase() просто выравнивает битовые поля дескриптора Base1, Base2 и Base3 так, чтобы получился 32-битный линейный адрес.

Листинг 4.17. Получение значений дескриптора

```
BOOL SpyDescriptor (PX86_SELECTOR      pSelector,
                    PX86_DESCRIPTOR    pDescriptor)
     {
     X86_SELECTOR        ldt;
     X86_TABLE           gdt;
     DWORD               dType, dLimit;
     BOOL                fSystem;
     PX86_DESCRIPTOR     pDescriptors  = NULL;
     BOOL                fOk           = FALSE;

     if (pDescriptor != NULL)
          {
          if (pSelector != NULL)
               {
               if (pSelector->TI) // дескриптор LDT
                    {
                    __asm
                         {
                         sldt ldt.wValue
                         sgdt gdt.wLimit
                         }
                    if ((!ldt.TI) && ldt.Index &&
                         ((ldt.wValue & X86_SELECTOR_INDEX)
                              <= gdt.wLimit))
                         {
                         dType  = SpyDescriptorType (
                                        gdt.pDescriptors + ldt.Index,
                                             &fSystem);
```

продолжение

Листинг 4.17 (*продолжение*)

```
                    dLimit = SpyDescriptorLimit (
                                gdt.pDescriptors + ldt.Index);

                    if ((dType == X86_DESCRIPTOR_SYS_LDT)[1]
                        &&
                        ((DWORD) (pSelector->wValue
                                    & X86_SELECTOR_INDEX)
                            <= dLimit))
                        {
                        pDescriptors = SpyDescriptorBase (
                                gdt.pDescriptors + ldt.Index);
                        }
                    }
                }
            else // дескриптор GDT
                {
                if (pSelector->Index)
                    {
                    __asm
                        {
                        sgdt gdt.wLimit
                        }
                    if ((pSelector->wValue  &
                          X86_SELECTOR_INDEX)
                        <= gdt.wLimit)
                        {
                        pDescriptors = gdt.pDescriptors;
                        }
                    }
                }
            }
        if (pDescriptors != NULL)
            {
            RtlCopyMemory  (pDescriptor,
                            pDescriptors + pSelector->Index,
                                X86_DESCRIPTOR_);
            fOk = TRUE;
            }
        else
            {
            RtlZeroMemory  (pDescriptor,
                                X86_DESCRIPTOR_);
            }
        }
    return fOk;
    }

//  ----------------------------------------+----------------

PVOID SpyDescriptorBase (PX86_DESCRIPTOR pDescriptor)
    {
    return (PVOID) ((pDescriptor->Base1          ) |
                    (pDescriptor->Base2 << 16) |
                    (pDescriptor->Base3 << 24));
    }
```

[1] В коде на компакт-диске в этом условии проверяется еще один операнд && : if (fSystem && (dType == ...). — *Примеч. перев.*

```
//  -------------------------------------------------------

DWORD SpyDescriptorLimit (PX86_DESCRIPTOR pDescriptor)
      {
      return (pDescriptor->G ? (pDescriptor->Limit1 << 12) |
                                        (pDescriptor->Limit2 << 28) |
                                        0xFFF
                                      : (pDescriptor->Limit1       ) |
                                        (pDescriptor->Limit2 << 16));
      }

//  -------------------------------------------------------

DWORD SpyDescriptorType (PX86_DESCRIPTORpDescriptor,
                                    PBOOL                    pfSystem)
      {
      if (pfSystem != NULL) *pfSystem = !pDescriptor->S;
      return pDescriptor->Type;
      }
```

Если бит TI селектора идентифицирует дескриптор GDT, все значительно упрощается. Снова инструкция SGDT используется для получения размера и месторасположения GDT в линейной памяти, и если указанный селектором индекс дескриптора находится в правильном диапазоне, переменной pDescriptors присваивается значение, указывающее на базовый адрес GDT. В обоих случаях, LDT и GDT, указатель pDescriptor будет ненулевым, если вызывающая функция передала допустимый селектор. В этом случае 64-битное значение дескриптора копируется в структуру вызывающей функции X86_DESCRIPTOR. В другом случае все члены этой структуры устанавливаются в ноль при помощи функции RtlZeroMemory().

Мы все еше обсуждаем функцию SpySegment(), показанную в листинге 4.15. Вызовы SpySelector() и SpyDescriptor() уже разобраны. Осталось рассмотреть только завершающие подпрограммы SpyDescriptorBase() и SpyDescriptorLimit(), но вы уже знаете, что делают эти функции (см. листинг 4.17). Если SpySelector() и SpyDescriptor() выполнятся успешно, то возвращаемые в структуре SPY_SEGMENT данные корректны. SpyDescriptorBase() и SpyDescriptorLimit() не возвращают флагов ошибок, потому что они не могут завершиться неудачно — они просто могут вернуть бессмысленные данные, если предоставленный им дескриптор некорректен.

Функция IOCTL SPY_IO_INTERRUPT

SPY_IO_INTERRUPT аналогична SPY_IO_SEGMENT, за исключением того, что эта функция работает с дескрипторами прерываний, хранимыми в системной таблице дескрипторов прерываний (Interrupt Descriptor Table, IDT), а не с дескрипторами таблицы LDT или GDT. В таблице IDT может содержаться до 256 дескрипторов, которые могут представлять шлюзы задачи, прерывания или ловушки (см. Intel, 1999c, pp. 5–11ff). Между прочим, природа прерываний и ловушек весьма схожа, различаясь только в одной небольшой детали: во время входа в обработчик прерывания прерывания всегда запрещены, в то время как при входе в обработчик ловушки флаг прерывания не изменяется. Номер прерывания в диапазоне с 0 по

256 содержится во входном буфере вызывающей функции SPY_IO_INTERRUPT, роль выходного буфера играет структура SPY_INTERRUPT, в которую при успешном возврате будут записаны свойства соответствующего обработчика прерываний. Вызываемая диспетчером вспомогательная функция-оболочка SpyOutputInterrupt() просто, в свою очередь, вызывает функцию SpyInterrupt() и копирует возвращаемые данные в выходной буфер. В листинге 4.18 показаны обе эти функции и структура SPY_INTERRUPT, поля которой заполняются SpyInterrupt() следующими элементами:

- Selector определяет селектор сегмента состояния задачи (Task State Segment, TSS, см. Intel, 1999c, pp. 6–4ff) или сегмент кода. Селектор сегмента кода указывает на сегмент, в котором расположены обработчик прерывания или ловушки;
- Gate — это 64-битный дескриптор шлюзов задачи, прерывания или ловушки; на который указывает селектор;
- Segment содержит свойства сегмента, на который указывает шлюз;
- pOffset определяет смещение точки входа обработчика прерывания или ловушки по отношению к базовому адресу содержащего обработчик сегмента кода. Поскольку в шлюзах задачи не содержится значение смещения, этот член должен быть пропущен, если входной селектор ссылается на TSS;
- fOK — это флаг корректности данных в структуре SPY_INTERRUPT.

Типичное использование TSS — обеспечить обработку ошибки нужной задачей. Это специальный тип системного сегмента, содержащий 104 байта информации о состоянии процессора, которая необходима для восстановления задачи после переключения задач, как показано в табл. 4.3. Когда происходит связанное с TSS прерывание, процессор всегда переключает задачу и сохраняет все регистры процессора в TSS. Windows 2000 хранит шлюз задачи в позициях прерываний 0x02 (немаскируемое прерывание, Nonmaskable Interrupt, NMI), 0x08 (двойная ошибка, Double Fault) и 0x12 (ошибка сегмента стека, Stack Segment Fault). Остальные записи указывают на обработчики прерываний. Неиспользуемые прерывания обрабатываются пустыми подпрограммами, называющимися KiUnexpectedInterruptNNN(), где NNN — это десятичный порядковый номер. Эти обработчики вызывают внутреннюю функцию KiEndUnexpectedRange(), которая, в свою очередь, вызывает KiUnexpectedInterruptTail(), передавая ей номер необрабатываемого прерывания.

Листинг 4.18. Запрос о свойствах прерывания

```
typedef struct _SPY_INTERRUPT
      {
      X86_SELECTOR   Selector;
      X86_GATE       Gate;
      SPY_SEGMENT    Segment;
      PVOID          pOffset;
      BOOL           fOk;
      }
      SPY_INTERRUPT, *PSPY_INTERRUPT, **PPSPY_INTERRUPT;

#define SPY_INTERRUPT_ sizeof (SPY_INTERRUPT)
```

```
//  ---------------------------------------------------------

NTSTATUS SpyOutputInterrupt (DWORD   dInterrupt,
                             PVOID   pOutput,
                             DWORD   dOutput,
                             PDWORD  pdInfo)
      {
      SPY_INTERRUPT si;

      SpyInterrupt (dInterrupt, &si);

      return SpyOutputBinary (&si, SPY_INTERRUPT_,
                              pOutput, dOutput, pdInfo);
      }

//  ---------------------------------------------------------

OOL SpyInterrupt (DWORD           dInterrupt,
                  PSPY_INTERRUPT  pInterrupt)
      {
      BOOL fOk = FALSE;

      if (pInterrupt != NULL)
           {
           if (dInterrupt <= X86_SELECTOR_LIMIT)
                {
                fOk = TRUE;

                if (!SpySelector (X86_SEGMENT_OTHER,
                                  dInterrupt << X86_SELECTOR_SHIFT,
                                            &pInterrupt->Selector))
                     {
                     fOk = FALSE;
                     }
                if (!SpyIdtGate  (&pInterrupt->Selector,
                                            &pInterrupt->Gate))
                     {
                     fOk = FALSE;
                     }
                if (!SpySegment  (X86_SEGMENT_OTHER,
                                            pInterrupt->Gate.Selector,
                                            &pInterrupt->Segment))
                     {
                     fOk = FALSE;
                     }
                pInterrupt->pOffset = SpyGateOffset (
                                                  &pInterrupt->Gate);
                }
           else
                {
                RtlZeroMemory (pInterrupt, SPY_INTERRUPT_);
                }
           pInterrupt->fOk = fOk;
           }
      return fOk;
      }
```

продолжение

Листинг 4.18 (*продолжение*)

```
//  ----------------------------------------------------

PVOID SpyGateOffset (PX86_GATE pGate)
      {
      return (PVOID) (pGate->Offset1 |
                           (pGate->Offset2 << 16));
 }
```

Таблица 4.3. Поля состояния процессора в сегменте состояния задачи (TSS)

Смещение	Биты	ID	Описание
0x00	16		Связь с предыдущей задачей
0x04	32	ESP0	Регистр-указатель стека (Stack Pointer) для уровня привилегий 0
0x08	16	SS0	Регистр сегмента стека (Stack Segment) для уровня привилегий 0
0x0C	32	ESP1	Регистр-указатель стека для уровня привилегий 1
0x10	16	SS1	Регистр сегмента стека для уровня привилегий 1
0x14	32	ESP2	Регистр-указатель стека для уровня привилегий 2
0x18	16	SS2	Регистр сегмента стека для уровня привилегий 2
0x1C	32	CR3	Базовый регистр каталога страниц (Page Directory Base Register, PDBR)
0x20	32	EIP	Регистр указателя команд (Instruction Pointer Register)
0x24	32	EFLAGS	Регистр флагов
0x28	32	EAX	Регистр общего назначения EAX
0x2C	32	ECX	Регистр общего назначения ECX
0x30	32	EDX	Регистр общего назначения EDX
0x34	32	EBX	Регистр общего назначения EBX
0x38	32	ESP	Регистр-указатель стека
0x3C	32	EBP	Регистр-указатель базы (Base Pointer)
0x40	32	ESI	Регистр-индекс источника (Source Index)
0x44	32	EDI	Регистр-индекс приемника (Destination Index)
0x48	16	ES	Регистр дополнительного сегмента (Extra Segment)
0x4C	16	CS	Регистр сегмента кода (Code Segment)
0x50	16	SS	Регистр сегмента стека
0x54	16	DS	Регистр сегмента данных (Data Segment)
0x58	16	FS	Дополнительный регистр сегмента данных № 1
0x5C	16	GS	Дополнительный регистр сегмента данных № 2
0x60	16	LDT	Селектор локальной таблицы дескрипторов (Local Descriptor Table, LDT)
0x64	1	T	Отладочный флаг системного прерывания (Trap)
0x66	16		Отображение базового адреса ввода-вывода
0x68	—		Окончание информации о состоянии процессора

Вызываемые SpyInterrupt() функции SpySegment() и SpySelector() были уже представлены в листингах 4.15 и 4.16. Функция SpyGateOffset(), расположенная в конце листинга 4.18, работает аналогично функциям SpyDescriptorBase() и SpyDescriptorLimit(), собирая битовые поля Offset1 и Offset2 структуры X86_GATE и выравнивая их соответствующим образом, формируя 32-битный адрес. SpyIdtGate() определена в листинге 4.19. Она будет сильно похожа на функцию SpyDescriptor() из листинга 4.17, если в по-

следней опустить работающий с таблицей LDT оператор. Ассемблерная инструкция SIDT сохраняет 48-битное содержимое регистра IDT, объединяя 16-битную границу таблицы и 32-битный линейный базовый адрес таблицы IDT. В остальном коде листинга 4.19 индекс дескриптора предоставленного селектора сравнивается с границей IDT, и если значение допустимо, соответствующий дескриптор прерывания копируется в структуру вызывающей функции X86_GATE, иначе все соответствующие шлюзам члены структуры устанавливаются в ноль.

Листинг 4.19. Получение значений шлюзов IDT

```
BOOL SpyIdtGate (PX86_SELECTOR  pSelector,
                 PX86_GATE      pGate)
      {
      X86_TABLE idt;
      PX86_GATE pGates    = NULL;
      BOOL      fOk       = FALSE;

      if (pGate != NULL)
           {
           if (pSelector != NULL)
                {
                __asm
                     {
                     sidt idt.wLimit
                     }
                if ((pSelector->wValue & X86_SELECTOR_INDEX)
                     <= idt.wLimit)
                     {
                     pGates = idt.pGates;
                     }
                }
           if (pGates != NULL)
                {
                RtlCopyMemory (pGate,
                               pGates + pSelector->Index,
                               X86_GATE_);
                fOk = TRUE;
                }
           else
                {
                RtlZeroMemory (pGate, X86_GATE_);
                }
           }
      return fOk;
      }
```

Функция IOCTL SPY_IO_PHYSICAL

Функция IOCTL SPY_IO_PHYSICAL организована достаточно просто, поскольку она полностью основывается на функции MmGetPhysicalAddress(), экспортируемой модулем ntoskrnl.exe. Обработчик функции IOCTL сначала получает линейный адрес при помощи функции SpyInputPointer() (см. листинг 4.10), затем вызывает функцию MmGetPhysicalAddress(), которая осуществляет поиск соответствующего физического адреса, и возвращает полученное значение PHYSICAL_ADDRESS. Заметьте, что тип

данных PHYSICAL_ADDRESS — LARGE_INTEGER. В большинстве систем i386 старшие 32 бита всегда будут равны нулю. Однако эти биты могут принимать ненулевые значения в системах с включенным режимом расширения физических адресов (Physical Address Extension, PAE) и объемом установленной памяти, большим 4 Гбайт.

Функция MmGetPhysicalAddress() для поиска физического адреса пользуется массивом PTE, который начинается с линейного адреса 0xC0000000. Основная схема преобразования работает следующим образом:

- если линейный адрес находится в диапазоне с 0x80000000 по 0x9FFFFFFF, то три самых значащих бита устанавливаются в ноль, что дает физический адрес в диапазоне с 0x00000000 до 0x1FFFFFFF;
- в другом случае старшие 20 бит используются в качестве индекса в массиве PTE, находящемся по адресу 0xC0000000;
- если в полученной записи PTE установлен бит P, означая наличие соответствующей страницы в памяти, все биты PTE, кроме двадцати бит PFN, сбрасываются, и к полученному значению добавляются младшие 12 бит линейного адреса, что дает нужный 32-битный физический адрес;
- если страница отсутствует в физической памяти, MmGetPhysicalAddress() возвращает ноль.

Интересно отметить, что функция MmGetPhysicalAddress() по умолчанию считает, что всем линейным адресам вне пределов диапазона адресов памяти ядра 0x80000000–0x9FFFFFFF соответствуют страницы объемом 4 Кбайт. Другие функции, такие как MmIsAddressValid(), сначала загружают запись PDE линейного адреса и проверяют ее бит PS, чтобы определить размер страницы (4 Кбайт или 4 Мбайт). Это гораздо более общий подход, который годится для любых конфигураций памяти. Обе функции возвращают верные значения, поскольку Windows 2000 использует страницы размером 4 Мбайт только в области памяти 0x80000000–0x9FFFFFFF. Вероятно, при проектировании одни функции API ядра были сделаны гибче других.

Функция IOCTL SPY_IO_CPU_INFO

Некоторые инструкции процессора доступны только программам, выполняющимся на самом высоком, нулевом уровне привилегий. В терминологии Windows 2000 это режим ядра. Среди запрещенных в том числе и инструкции, читающие содержимое управляющих регистров CR0, CR1 и CR3. Поскольку в этих регистрах хранится весьма интересная информация, может возникнуть желание получить к ним доступ, что и реализует функция SPY_IO_CPU_INFO. Как показано в листинге 4.20, вызываемая обработчиком IOCTL функция SpyOutputCpuInfo() при помощи встроенного кода на языке ассемблера читает значения управляющих регистров, а также другую полезную информацию, такую как содержимое регистров таблиц IDT, GDT и LDT и значения регистров CS, DS, ES, FS, GS, SS и TR, в которых хранятся селекторы сегментов. В регистре задачи (Task Register) TR хранится селектор, указывающий на сегмент TSS текущей задачи.

Листинг 4.20. Получение информации о состоянии процессора

```
typedef struct _SPY_CPU_INFO
      {
      X86_REGISTER   cr0;
      X86_REGISTER   cr2;
      X86_REGISTER   cr3;
      SPY_SEGMENT    cs;
      SPY_SEGMENT    ds;
      SPY_SEGMENT    es;
      SPY_SEGMENT    fs;
      SPY_SEGMENT    gs;
      SPY_SEGMENT    ss;
      SPY_SEGMENT    tss;
      X86_TABLE      idt;
      X86_TABLE      gdt;
      X86_SELECTOR   ldt;
      }
      SPY_CPU_INFO, *PSPY_CPU_INFO, **PPSPY_CPU_INFO;

#define SPY_CPU_INFO_ sizeof (SPY_CPU_INFO)

//  ------------------------------------------------------

NTSTATUS SpyOutputCpuInfo (PVOID    pOutput,
                           DWORD    dOutput,
                           PDWORD   pdInfo)
      {
      SPY_CPU_INFO  sci;
      PSPY_CPU_INFO psci = &sci;

      __asm
          {
          push      eax
          push      ebx
          mov       ebx, psci

          mov       eax, cr0
          mov       [ebx.cr0], eax

          mov       eax, cr2
          mov       [ebx.cr2], eax

          mov       eax, cr3
          mov       [ebx.cr3], eax

          sidt      [ebx.idt.wLimit]
          mov       [ebx.idt.wReserved], 0

          sgdt      [ebx.gdt.wLimit]
          mov       [ebx.gdt.wReserved], 0

          sldt      [ebx.ldt.wValue]
          mov       [ebx.ldt.wReserved], 0

          pop       ebx
          pop       eax
          }
```

продолжение

Листинг 4.20 (*продолжение*)

```
    SpySegment (X86_SEGMENT_CS,  0, &sci.cs);
    SpySegment (X86_SEGMENT_DS,  0, &sci.ds);
    SpySegment (X86_SEGMENT_ES,  0, &sci.es);
    SpySegment (X86_SEGMENT_FS,  0, &sci.fs);
    SpySegment (X86_SEGMENT_GS,  0, &sci.gs);
    SpySegment (X86_SEGMENT_SS,  0, &sci.ss);
    SpySegment (X86_SEGMENT_TSS, 0, &sci.tss);

    return SpyOutputBinary (&sci, SPY_CPU_INFO_,
                            pOutput, dOutput, pdInfo);
    }
```

Селекторы сегментов извлекаются при помощи функции SpySegment(), обсуждаемой ранее (подробнее см. листинг 4.15).

Функция IOCTL SPY_IO_PDE_ARRAY

SPY_IO_PDE_ARRAY — еще одна тривиальная функция, которая просто копирует каталог страниц целиком с адреса 0xC0300000 в выходной буфер вызывающей функции. Буфер должен быть представлен в виде структуры SPY_PDE_ARRAY, показанной в листинге 4.21. Как вы уже догадались, размер этой структуры в точности равен 4 Кбайт и она состоит из 1024 32-битных значений PDE. Здесь используется представляющая обобщенную запись страницы структура X86_PE, ее описание можно найти в листинге 4.3, а константа X86_PAGES_4M определена в листинге 4.5. Поскольку массив SPY_PDE_ARRAY всегда содержит в качестве элементов записи каталога страниц, встроенные структуры X86_PE принадлежат либо к типу X86_PDE_4M, либо к типу X86_PDE_4K, что определяется значением бита размера страницы PS.

Как правило, копирование содержимого памяти без проверки наличия страницы-источника в физической памяти не приветствуется. Тем не менее каталог страниц — одно из немногих исключений. Каталог страниц текущей задачи всегда находится в физической памяти во время ее выполнения. Он не может быть выгружен в файл подкачки до тех пор, пока не произойдет переключения на другую задачу. Именно поэтому в базовом регистре каталога страниц (Page Directory Base Register, PDBR) нет бита *P* (present, присутствие), который есть в регистрах таблиц PDE и PTE, как показано в листинге 4.3 в определении структуры X86_PDBR.

Листинг 4.21. Определение структуры SPY_PDE_ARRAY

```
typedef struct _SPY_PDE_ARRAY
    {
    X86_PE apde [X86_PAGES_4M];
    }
    SPY_PDE_ARRAY, *PSPY_PDE_ARRAY, **PPSPY_PDE_ARRAY;

#define SPY_POE_ARRAY_ sizeof (SPY_PDE_ARRAY)
```

Функция IOCTL SPY_IO_PAGE_ENTRY

Эта функция удобна, когда нужно получить доступ к записи страницы по заданному линейному адресу. В листинге 4.22 показано определение функции обрабатывающей этот запрос IOCTL. Возвращаемая структура SPY_PAGE_ENTRY по большей части аналогична записи страницы X86_PE, обладая еще двумя удобными дополнениями: член dSize задает размер страницы в байтах, принимая одно из двух значений: либо X86_PAGE_4K (4096 байт), либо X86_PAGE_4M (4 194 304 байт); и член fPresent показывает, находится ли страница в физической памяти. Этот флаг следует отличать от самого возвращаемого функцией SpyMemoryPageEntry() значения, которое может равняться TRUE, даже если fPresent равно FALSE. В этом случае переданный линейный адрес корректен, но указывает на страницу, в данный момент выгруженную в файл подкачки. В такой ситуации установлен бит № 10 записи страницы, в листинге 4.22 названный PageFile, а бит P — нет. Дополнительные детали рассмотрены при описании структуры X86_PNE ранее в этой главе. Структура X86_PNPE представляет запись для отсутствующей в физической памяти страницы, ее определение приведено в листинге 4.3.

Функция SpyMemoryPageEntry() предполагает, что размер запрашиваемой страницы 4 Мбайт, и копирует запись PDE указанного линейного адреса из массива PDE системы, расположенного по адресу 0xC0300000, в член pe структуры SPY_PAGE_ENTRY. Если бит P установлен, присутствует либо страница нижнего уровня, либо таблица страниц, поэтому на следующем шаге проверяется бит PS размера страницы. Если он установлен, запись PDE указывает на страницу размером 4 Мбайт и функция заканчивает работу — SpyMemoryPageEntry() возвращает TRUE и член fPresent структуры SPY_PAGE_ENTRY также устанавливается в TRUE. Если бит PS равен нулю, запись PDE ссылается на таблицу PTE, поэтому код извлекает эту таблицу из массива, расположенного по адресу 0xC0000000, и проверяет его бит P. Если бит установлен, содержащая линейный адрес страница объемом 4 Кбайт находится в физической памяти, член fPresent устанавливается в TRUE и функция SpyMemoryPageEntry() также возвращает TRUE. В ином случае извлеченное значение должно быть записью для отсутствующей страницы, поэтому fPresent устанавливается в FALSE, а функция SpyMemoryPageEntry() возвращает TRUE, только когда установлен бит PageFile записи страницы.

Листинг 4.22. Извлечение записей PDE и PTE

```
typedef struct _SPY_PAGE_ENTRY
      {
      X86_PE pe;
      DWORD  dSize;
      BOOL fPresent;
      }
      SPY_PAGE_ENTRY, *PSPY_PAGE_ENTRY, **PPSPY_PAGE_ENTRY;

#define SPY_PAGE_ENTRY_ sizeof (SPY_PAGE_ENTRY)

//  ------------------------------------------------------
```

продолжение

Листинг 4.22 (*продолжение*)

```
BOOL SpyMemoryPageEntry (PVOID            pVirtual,
                         PSPY_PAGE_ENTRY  pspe)
    {
    SPY_PAGE_ENTRY spe;
    BOOL           fOk = FALSE;

    spe.pe       = X86_PDE_ARRAY [X86_PDI (pVirtual)];
    spe.dSize    = X86_PAGE_4M;
    spe.fPresent = FALSE;

    if (spe.pe.pde4M.P)
        {
        if (spe.pe.pde4M.PS)
            {
            fOk = spe.fPresent = TRUE;
            }
        else
            {
            spe.pe     = X86_PTE_ARRAY [X86_PAGE (pVirtual)];
            spe.dSize  = X86_PAGE_4K;

            if (spe.pe.pte4K.P)
                {
                fOk = spe.fPresent = TRUE;
                }
            else
                {
                fOk = (spe.pe.pnpe.PageFile != 0);
                }
            }
        }
    if (pspe != NULL) *pspe = spe;
    return fOk;
    }
```

Заметьте, что функция SpyMemoryPageEntry() не распознает выгруженные страницы объемом 4 Мбайт. Если запись PDE для страницы 4 Мбайт указывает на отсутствующую страницу, никак нельзя узнать, что это вызвано некорректностью линейного адреса или тем, что страница выгружена в файл подкачки. Страницы объемом 4 Мбайт используются только в диапазоне адресов памяти ядра 0x80000000–0x9FFFFFFF. Я еще никогда не сталкивался с ситуацией, даже при острой нехватки памяти, в которой эти страницы выгружались бы из памяти, поэтому у меня не было возможности исследовать какие-либо записи для отсутствующей страницы.

Функция IOCTL SPY_IO_MEMORY_DATA

Функция IOCTL SPY_IO_MEMORY_DATA, безусловно, одна из наиболее важных, потому что она копирует произвольное количество данных из памяти в буфер вызывающей программы. Как вы, возможно, помните, приложения пользовательского режима зачастую передают неверные адреса. Следовательно, эта функция очень осторожна и проверяет допустимость всех исходных адресов до того, как читать их значение. Не забывайте, что в режиме ядра синий экран смерти готов показаться в любую минуту.

Вызывающее приложение запрашивает содержимое блока памяти, передавая задающую его адрес и размер структуру SPY_MEMORY_BLOCK, показанную в верхней части листинга 4.23. Для удобства адрес определен как объединение, допуская интерпретацию в виде массива байт (PBYTE pbAddress) или в виде произвольного указателя (PVOID pAddress). Функция SpyInputMemory() в листинге 4.23 копирует эту структуру из входного буфера IOCTL. Ее функция-напарник SpyOutputMemory(), приведенная в листинге 4.23, является оболочкой функции SpyMemoryReadBlock(), показанной в листинге 4.24. Основная обязанность SpyOutputMemory() — вернуть соответствующие значения NTSTATUS, в то время как SpyMemoryReadBlock() обеспечивает данные.

Функция SpyMemoryReadBlock() возвращает содержимое памяти в структуре SPY_MEMORY_DATA, определенной в листинге 4.25. Здесь я применил другой подход, чем в предыдущих определениях, потому что структура SPY_MEMORY_DATA представляет собой тип данных переменного размера. По существу, она состоит из структуры SPY_MEMORY_BLOCK с именем smb и последующим массивом awData[] элементов DWORD, длина которого определяется членом dBytes структуры smb. Чтобы было возможно легко определять экземпляры SPY_MEMORY_DATA в виде глобальных или локальных переменных предопределенного размера, определение структуры основывается на макросе SPY_MEMORY_DATA_N(), единственный аргумент которого задает размер массива awData[]. Фактическое определение структуры следует за определением макроса, определяя SPY_MEMORY_DATA с массивом awData[] нулевой длины. Макрос SPY_MEMORY_DATA__() вычисляет общий размер структуры SPY_MEMORY_DATA по заданному размеру ее массива данных, а остальные определения реализуют упаковку и распаковку данных типа WORD в массиве. Понятно, что младшая половина каждого элемента типа WORD содержит байт данных памяти, а старшая половина определяет флаги. Сейчас имеет значение только бит № 8, показывающий корректность байта данных, расположенного в битах с № 0 по № 7.

Листинг 4.23. Работа с блоками памяти

```
typedef struct _SPY_MEMORY_BLOCK
      {
      union
           {
           PBYTE pbAddress;
           PVOID pAddress;
           };
      DWORD dBytes;
      }
      SPY_MEMORY_BLOCK,  *PSPY_MEMORY_BLOCK,
                              **PPSPY_MEMORY_BLOCK;

#define SPY_MEMORY_BLOCK_ sizeof (SPY_MEMORY_BLOCK)

//  ------------------------------------------------------

NTSTATUS SpyInputMemory (PSPY_MEMORY_BLOCK   psmb,
                         PVOID               pInput,
                         DWORD               dInput)
```

продолжение

Листинг 4.23 (*продолжение*)

```
    {
    return SpyInputBinary (psmb, SPY_MEMORY_BLOCK_,
                           pInput, dInput);
    }

//  ------------------------------------------------------

NTSTATUS SpyOutputMemory (PSPY_MEMORY_BLOCK   psmb,
                          PVOID               pOutput,
                          DWORD               dOutput,
                          PDWORD              pdInfo)
    {
    NTSTATUS ns = STATUS_BUFFER_TOO_SMALL;

    if (*pdInfo = SpyMemoryReadBlock (
                                  psmb, pOutput, dOutput))
        {
        ns = STATUS_SUCCESS;
        }
    return ns;
    }
```

Листинг 4.24. Копирование содержимого блока памяти

```
DWORD SpyMemoryReadBlock (PSPY_MEMORY_BLOCK   psmb,
                          PSPY_MEMORY_DATA    psmd,
                          DWORD               dSize)
    {
    DWORD i;
    DWORD n = SPY_MEMORY_DATA__ (psmb->dBytes);

    if (dSize >= n)
        {
        psmd->smb = *psmb;

        for (i = 0; i < psmb->dBytes; i++)
            {
            psmd->awData [i] =
                (SpyMemoryTestAddress (psmb->pbAddress + i)
                 ? SPY_MEMORY_DATA_VALUE (
                                  psmb->pbAddress [i], TRUE)
                 : SPY_MEMORY_DATA_VALUE (0, FALSE));
            }
        }
    else
        {
        if (dSize >= SPY_MEMORY_DATA_)
            {
            psmd->smb.pbAddress = NULL;
            psmd->smb.dBytes    = 0;
            }
        n = 0;
        }
    return n;
    }
```

```
//  ------------------------------------------------------

BOOL SpyMemoryTestAddress (PVOID pVirtual)
      {
      return SpyMemoryPageEntry (pVirtual, NULL);
      }

//  ------------------------------------------------------

BOOL SpyMemoryTestBlock (PVOID  pVirtual,
                         DWORD  dBytes)
      {
      PBYTE pbData;
      DWORD dData;
      BOOL  fOk = TRUE;

      if (dBytes)
           {
           pbData = (PBYTE) ((DWORD_PTR) pVirtual &
                                                  X86_PAGE_MASK);
           dData  = (((dBytes + X86_OFFSET_4K (pVirtual) - 1)
                           / PAGE_SIZE) + 1) * PAGE_SIZE;
           do  {
                    fOk = SpyMemoryTestAddress (pbData);
                    pbData += PAGE_SIZE;
                    dData  -= PAGE_SIZE;
                    }
           while (fOk && dData);
           }
      return fOk;
      }
```

Проверка байта данных на допустимость осуществляется функцией SpyMemoryTestAddress(), которая вызывается функцией SpyMemoryReadBlock() для адреса каждого байта перед копированием его в буфер. SpyMemoryTestAddress(), показанная во второй половине листинга 4.24, в свою очередь, вызывает SpyMemoryPageEntry(), устанавливая ее второй аргумент в NULL. Последнюю функцию мы только что рассмотрели во время обсуждения функции IOCTL SPY_IO_PAGE_ENTRY (листинг 4.22). Значение NULL ее аргумента-указателя PSPY_PAGE_ENTRY означает, что вызывающую функцию не интересует запись страницы для переданного линейного адреса, поэтому учитывается только возвращаемое значение функции, в случае допустимости линейного адреса равное TRUE. Функция SpyMemoryPageEntry() считает адрес допустимым, если содержащая его страница находится либо в физической памяти, либо в одном из файлов подкачки системы. Заметьте, что эта трактовка отличается от трактовки расположенной в модуле ntoskrnl.exe функции API MmIsAddressValid(), которая всегда возвращает FALSE при отсутствии страницы в физической памяти, даже если это корректная страница, в данный момент расположенная в файле подкачки. Расширенной версией функции SpyMemoryTestAddress() является функция SpyMemoryTestBlock(), также представленная в листинге 4.24. Она проверяет заданный блок памяти на допустимость участками по 4096 байт, проверяя на допустимость каждую страницу участка.

Листинг 4.25. Определение структуры SPY_MEMORY_DATA

```
#define SPY_MEMORY_DATA_N(_n) \
          struct _SPY_MEMORY_DATA_##_n \
               { \
               SPY_MEMORY_BLOCK    smb; \
               WORD                awData [_n]; \
               }

typedef SPY_MEMORY_DATA_N (0)
          SPY_MEMORY_DATA,  *PSPY_MEMORY_DATA,
                                **PPSPY_MEMORY_DATA;

#define SPY_MEMORY_DATA_ sizeof (SPY_MEMORY_DATA)
#define SPY_MEMORY_DATA_(_n) \
                    (SPY_MEMORY_DATA_ + ((_n) * WORD_))

#define SPY_MEMORY_DATA_BYTE   0x00FF
#define SPY_MEMORY_DATA_VALID  0x0100

#define SPY_MEMORY_DATA_VALUE(_b,_v) \
               ((WORD) (((_b) & SPY_MEMORY_DATA_BYTE       ) | \
                              ((_v) ? SPY_MEMORY_DATA_VALID : 0)))
```

Рассмотрение диапазонов адресов выгруженных страниц в качестве допустимых обладает тем важным преимуществом, что страница будет загружена обратно в физическую память сразу после того, как SpyMemoryReadBlock() попытается обратиться к одному из ее байт. Представленная далее утилита чтения памяти не представляла бы особого интереса, если бы полагалась в своей работе на MmIsAddressValid(). Иногда не отображалось бы содержимое определенных диапазонов адресов, хотя только что это можно было сделать, поскольку соответствующая им страница была перемещена в файл подкачки.

Функция IOCTL SPY_IO_MEMORY_BLOCK

Функция IOCTL SPY_IO_MEMORY_BLOCK похожа на SPY_IO_MEMORY_DATA, она также копирует блоки даных, расположенные по произвольным адресам, в предоставленный буфер. Главное отличие заключается в том, что SPY_IO_MEMORY_DATA пытается копировать все доступные байты, а SPY_IO_MEMORY_BLOCK завершается неудачно, если в запрашиваемом диапазоне находится какой-либо некорректный адрес. Эта функция понадобится нам в главе 6 для передачи в приложение пользовательского режима содержимого структур данных, расположенных в памяти, отведенной ядру. Ясно, что такая функция должна ограничиваться только допустимыми данными — нельзя копировать структуру с недостижимыми байтами, поскольку в копии будет отсутствовать часть данных.

Аналогично SPY_IO_MEMORY_DATA, входные данные должны быть переданы функции SPY_IO_MEMORY_BLOCK в виде структуры SPY_MEMORY_BLOCK, задающей базовый адрес и размер копируемой области памяти. Данные копируются из памяти один к одному. Выходной буфер должен быть достаточно большим, чтобы вместить копию целиком, если это не так, возвращается только сообщение об ошибке, никакие данные обратно не передаются.

Функция IOCTL SPY_IO_HANDLE_INFO

Так же как и представленная выше функция SPY_IO_PHYSICAL, эта функция дает возможность приложению пользовательского режима обращаться к функциям API режима ядра, которые нельзя вызвать по-другому. Драйвер режима ядра всегда может получить указатель на объект по его дескриптору, вызвав функцию ObReferenceObjectByHandle() из модуля ntoskrnl.exe. В Win32 API нет эквивалентной функции. Однако драйвер слежения может вызвать функцию от своего имени и затем передать указатель на объект в приложение режима пользователя. В листинге 4.26 показано, как SpyDispatcher() получает на вход дескриптор объекта через функцию SpyInputHandle(), определенную в листинге 4.10, и затем вызывает функцию SpyOutputHandleInfo().

Структура SPY_HANDLE_INFO, приведенная в начале листинга 4.26, получает от функции ObReferenceObjectByHandle() указатель на тело сопоставленного дескриптору объекта и атрибуты дескриптора объекта. Важно отметить, что если ObReferenceObjectByHandle() завершится успешно, нужно вызвать ObDeReferenceObject(), для того чтобы установить счетчик ссылок объекта в его предыдущее значение. Если этого не сделать, возникнет «утечка ссылок объекта».

Листинг 4.26. Доступ к объекту по его дескриптору

```
typedef struct _SPY_HANDLE_INFO
      {
      PVOID pObjectBody;
      DWORD dHandleAttributes;
      }
         SPY_HANDLE_INFO,
        *PSPY_HANDLE_INFO,
       **PPSPY_HANDLE_INFO;

#define SPY_HANDLE_INFO_ sizeof (SPY_HANDLE_INFO)

//  ------------------------------------------------------

NTSTATUS SpyOutputHandleInfo (HANDLE    hObject,
                              PVOID     pOutput,
                              DWORD     dOutput,
                              PDWORD    pdInfo)
      {
      SPY_HANDLE_INFO              shi;
      OBJECT_HANDLE_INFORMATION    ohi;
      NTSTATUS                     ns = STATUS_INVALID_PARAMETER;

      if (hObject != NULL)
          {
          ns = ObReferenceObjectByHandle (hObject,
                                          STANDARD_RIGHTS_READ,
                                          NULL, KernelMode,
                                          &shi.pObjectBody,
                                          &ohi);
          }
      if (ns == STATUS_SUCCESS)
```

продолжение

Листинг 4.26 (*продолжение*)

```
        {
        shi.dHandleAttributes = ohi.HandleAttributes;

        ns = SpyOutputBinary (&shi, SPY_HANDLE_INFO_,
                                    , pOutput, dOutput, pdInfo);

        ObDereferenceObject (shi.pObjectBody);
        }
    return ns;
    }
```

Пример утилиты просмотра памяти

Проработав сложный и, возможно, запутанный код функций-обработчиков IOCTL драйвера слежения памяти, вы уже, наверное, хотите посмотреть на эти функции в действии. Для этого я создал консольное приложение «SBS Windows 2000 Memory Spy», загружающее драйвер слежения и вызывающее различные функции IOCTL в зависимости от параметров командной строки. Исполнимый код приложения расположен в файле w2k_mem.exe, а исходный код можно найти на прилагающемся к книге компакт-диске в каталоге \src\w2k_mem.

Формат командной строки

Утилиту просмотра памяти можно запустить с компакт-диска, выполнив команду d:\bin\w2k_mem.exe, где на месте d: должна стоять буква диска, соответствующая вашему накопителю компакт-дисков. Если запустить w2k_mem.exe без аргументов, на экране появится длинный список команд, как показано в примере 4.1. Основной принцип команд w2k_mem.exe состоит в том, что команда состоит из одного или более запросов данных, для каждого из запросов должен быть предоставлен по крайней мере линейный базовый адрес, с которого должен начинаться снимок памяти. Можно также указать размер блока памяти, если размер не задан, он по умолчанию считается равным 256. При указании размера памяти значению должен предшествовать символ «#». Схема работы команды по умолчанию может быть изменена при помощи нескольких ключей-параметров. Параметр состоит из односимвольного идентификатора параметра и префикса «+» или «-», соответственно включающего и выключающего параметр. По умолчанию все параметры отключены.

Пример 4.1. Справочный экран утилиты просмотра памяти

```
// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_mem { { [+option|-option] [/<path>] }[#[[0]x]<size>] [[0]x]<base> }

<path> specifies a module to be loaded into memory.
Use the +x/-x switch to enable/disable its startup code.
If <size> is missing, the default size is 256 bytes.
```

```
Display address options (mutually exclusive):

      +z -z     zero-based display          on / OFF
      +r -r     physical RAM addresses      on / OFF

Display mode options (mutually exclusive):

      +w -w     WORD  data formatting       on / OFF
      +d -d     DWORD data formatting       on / OFF
      +q -q     QWORD data formatting       on / OFF

Addressing options (mutually exclusive):

      +t -t     TEB-relative addressing     on / OFF
      +f -f     FS-relative  addressing     on / OFF
      +u -u     user-mode FS:[<base>]       on / OFF
      +k -k     kernel-mode FS:[<base>]     on / OFF
      +h -h     handle/object resolution    on / OFF
      +a -a     add bias  to  last base     on / OFF
      +s -s     sub bias from last base     on / OFF
      +p -p     pointer  from last block    on / OFF

System status options (cumulative):

      +o -o     display OS  information     on / OFF
      +c -c     display CPU information     on / OFF
      +g -g     display GOT information     on / OFF
      +i -i     display IDT information     on / OFF
      +b -b     display contiguous blocks   on / OFF

Other options (cumulative):

      +x -x     execute DLL startup code    on / OFF

Example: The following command displays the first 64
bytes of the current Process Environment Block (PEB)
in zero-based DWORD format. assuming that a pointer to
the PEB is located at offset 0x30 inside the current
Thread Environment Block (TEB):

      w2k_mem +t #0 0 +pzd #64 0x30

Note: Specifying #0 after +t causes the TEB to be
addressed without displaying its contents.
```

Запрос данных выполняется для каждой лексемы командной строки, которая не может быть идентифицирована как параметр, спецификация размера блока, путь или любой другой модификатор команды. Предполагается, что каждое обычное число в командной строке — это линейный адрес, и выводится шестнадцатеричный снимок памяти, начиная с этого адреса. Числа по умолчанию интерпретируются как десятичные, префикс «0x» или просто «x» задает шестнадцатеричное число.

Сложные схемы работы с командной строкой, наподобие реализованной в w2k_mem.exe, гораздо легче воспринимаются на основе примеров. Вот некоторые из них:

- w2k_mem 0x80400000 выводит первые 256 байт памяти, начиная с линейного адреса 0x80400000, при этом должно получиться что-то похожее на пример 4.2. Между прочим, это заголовок DOS модуля ntoskrnl.exe (обратите внимание на идентификатор «MZ» в начале);
- w2k_mem #0x40 0x80400000 отображает тот же блок данных, но останавливается после вывода 64 байт, как указано в спецификации размера блока #0x40;
- w2k_mem +d 0x40 0x80400000 еще один вариант, на этот раз байты упаковываются в 32-битные величины типа DWORD, как задает параметр +d. Этот параметр остается в силе до тех пор, пока он не будет отменен ключом –d или перекрыт взаимоисключающим параметром, таким как +w или +q;
- w2k_mem +wz #0x40 0x10000 +d –z 0x20000 содержит два запроса данных. Первый запрос определяет диапазон адресов 0x10000–0x1003F, показанных в формате 16-битных величин типа WORD, второй дает диапазон 0x20000–0x2003F в формате DWORD (пример 4.3). Первый запрос также содержит переключатель +z, в соответствии с которым числа в столбце «Address» начинаются с нуля. Для второго запроса режим отображения с нуля выключен параметром –z;
- w2k_mem +rd #4096 0xC0300000 отображает каталог страниц системы по адресу 0xC0300000 в формате DWORD. Параметр +r включает отображение в столбце «Address» адресов физической оперативной памяти вместо линейных адресов.

Теперь у вас уже должно сложиться представление о работе утилиты в командной строке. Специальные параметры и особенности будут более подробно освещены в следующих подразделах. Большая часть этих параметров различным образом влияет на интерпретацию адреса, заданного в следующем параметре. В режиме по умолчанию заданный адрес рассматривается как линейный базовый адрес, с которого начинается снимок памяти. С параметрами +t, +f, +u, +k, +h, +a, +s и +p адрес трактуется по-другому.

Пример 4.2. Образец запроса данных

```
E:\>w2k_mem 0x80400000

// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Loading "SBS Windows 2000 Spy Device" (w2k_spy) ...
Driver: "D:\Program Files\DevStudio\MyProjects\w2k_mem\
Release\w2k_spy.sys"
Opening "\\.\w2k_spy" ...

SBS Windows 2000 Spy Device V1.00 ready

80400000..804000FF: 256 valid bytes
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|-----------------
80400000 | 4D 5A 90 00-03 00 00 00 : 04 00 00 00-FF FF 00 00 | MZ•.........яя..
```

```
80400010 | B8 00 00 00-00 00 00 00 : 40 00 00 00-00 00 00 00 | ё.......@.......
80400020 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
80400030 | 00 00 00 00-00 00 00 00 : 00 00 00 00-C8 00 00 00 | ............И...
80400040 | 0E 1F BA 0E-00 B4 09 CD : 21 B8 01 4C-CD 21 54 68 | ..є..r.H!ё.LH!Th
80400050 | 69 73 20 70-72 6F 67 72 : 61 6D 20 63-61 6E 6E 6F | is program canno
80400060 | 74 20 62 65-20 72 75 6E : 20 69 6E 20-44 4F 53 20 | t be run in DOS
80400070 | 6D 6F 64 65-2E 0D 0D 0A : 24 00 00 00-00 00 00 00 | mode....$.......
80400080 | 50 7A C4 CE-14 1B AA 9D : 14 1B AA 9D-14 1B AA 9D | PzДО..Є•..Є•..Є•
80400090 | 14 1B AB 9D-53 1B AA 9D : 18 3B A4 9D-5B 1B AA 9D | ..«•S.Є•.;¤•[.Є•
804000A0 | 42 13 AC 9D-15 1B AA 9D : 14 1B AA 9D-1A 19 AA 9D | B.¬•..Є•..Є•..Є•
804000B0 | 4D 38 B9 9D-12 1B AA 9D : 52 69 63 68-14 1B AA 9D | M8№•..Є•Rich..Є•
804000C0 | 00 00 00 00-00 00 00 00 : 50 45 00 00-4C 01 13 00 | ........PE..L...
804000D0 | 17 9B 4D 38-00 00 00 00 : 00 00 00 00-E0 00 0E 03 | .?M8........a...
804000E0 | 0B 01 05 0C-C0 2D 14 00 : B0 D6 04 00-00 00 00 00 | ....A-..?O......
804000F0 | 20 D1 00 00-C0 04 00 00 : B0 73 06 00-00 00 40 00 |  N..A...?s....@.

          256 bytes requested
          256 bytes received

Closing the spy device ...
```

Пример 4.3. Отображение данных в специальных форматах

```
E:\>w2k_mem +wz #0x40 0x10000 +d -z 0x20000

// w2k_mem.exe
// SBS Windows 2000 Memory Spy V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Loading "SBS Windows 2000 Spy Device" (w2k_spy) ...
Driver: "D:\Program Files\DevStudio\MyProjects\w2k_mem\
Release\w2k_spy.sys"
Opening "\\.\w2k_spy" ...

SBS Windows 2000 Spy Device V1.00 ready

00010000..0001003F: 64 valid bytes

Address  | 0000 0002-0004 0006 : 0008 000A-000C 000E | 00 02 04 06 08 0A 0C 0E
---------|---------------------:---------------------|-----------------------
00000000 | 003D 0044-003A 003D : 0044 003A-005C 0050 | .= .D .: .= .D .: .\ .P
00000010 | 0072 006F-0067 0072 : 0061 006D-0020 0046 | .r .o .g .r .a .m .  .F
00000020 | 0069 006C-0065 0073 : 005C 0044-0065 0076 | .i .l .e .s .\ .D .e .v
00000030 | 0053 0074-0075 0064 : 0069 006F-005C 004D | .S .t .u .d .i .o .\ .M

00020000..0002003F: 64 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
00020000 | 00001000 - 00000880 : 00000001 - 00000000 | .... ...? .... ....
00020010 | 02B20001 - 00000000 : 00000003 - 00000007 | .I.. .... .... ....
00020020 | 0000000B - 0208006C : 00020290 - 00000018 | .... ...l ... ....
00020030 | 02A0029E - 00020498 : 008400B2 - 00020738 | . .? ...? .?.? ...8

          128 bytes requested
          128 bytes received

Closing the spy device ...
```

Относительная адресация через TEB

У каждого потока в процессе есть свой собственный блок переменных окружения потока (Thread Environment Block, TEB,), в котором система хранит постоянно используемые данные потока. В пользовательском режиме блок TEB текущего потока расположен в отдельном сегменте объемом 4 Кбайт, доступном через регистр процессора FS. В режиме ядра FS указывает на другой сегмент, как будет объяснено ниже. Все блоки TEB процесса кладутся в стек в линейной памяти по линейному адресу 0x7FFDE000, каждый новый блок расширяет стек вниз на 4 Кбайт. Таким образом, TEB второго потока расположен по адресу 0x7FFDD000, TEB третьего потока — по адресу 0x7FFDC000 и т. д. Содержимое блоков TEB и блок переменных окружения процесса (Process Environment Block, PEB), расположенный по адресу 0x7FFDF000, будут подробно обсуждаться в главе 7 (см. листинги 7.18 и 7.19). Для рассмотрения текущей темы достаточно отметить, что блоки TEB существуют и что на них указывает регистр FS.

Если в команде адресу предшествует параметр +t, w2k_mem.exe добавляет к адресу базовый адрес сегмента FS, что дает смещение в 0x7FFDE000 байт. В примере 4.4 показан результат выполнения команды w2k_mem +dt #0x38 0 в моей системе. На этот раз заголовок и сообщения о состоянии не приведены, опущенные части помечены [...].

Пример 4.4. Отображение первого блока переменных окружения потока (TEB)

```
E:\>w2k_mem +dt #0x38 0
[...]
7FFDE000..7FFDE037: 56 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
7FFDE000 | 0012FA58 - 00130000 : 0012E000 - 00000000 | ..ъX .... ..а. ....
7FFDE010 | 00001E00 - 00000000 : 7FFDE000 - 00000000 | .... .... .за. ....
7FFDE020 | 000002C0 - 000002C8 : 00000000 - 00000000 | ...А ...И .... ....
7FFDE030 | 7FFDF000 - 00000000 :           -         | .зр. ....
[...]
```

Относительная адресация через регистр FS

Как я уже говорил, в пользовательском режиме и режиме ядра регистр FS указывает на различные сегменты. Так же как ключ +t выбирает адрес FS режима пользователя за точку отсчета, параметр +f использует базовый адрес FS, действительный для режима ядра. Конечно, приложению Win32 это значение недоступно, поэтому нам снова понадобится драйвер слежения. w2k_mem.exe обращается к функции IOCTL SPY_IO_CPU_INFO, объясненной в предыдущей секции, чтобы прочитать информацию о состоянии процессора, включающую значения режима ядра всех сегментных регистров. После этого все происходит точно так же, как и с ключом +t.

Регистр FS в режиме ядра указывает на другую привязанную к потоку структуру, к которой часто обращается ядро Windows 2000, — управляющую область процессора (Kernel's Processor Region, KPCR). Мы уже рассматривали эту структуру в ходе обсуждения функции IOCTL SPI_IO_OS_INFO и снова к ней вернемся

в главе 7 (см. листинг 7.16). Опять же, сейчас нам достаточно знать, что эта структура существует и расположена по линейному адресу 0xFFDFF000 и что к ней легко можно получить доступ при помощи ключа +f. Пример 4.5 демонстрирует применения ключа +f, показывая, что при этом смещение в 0xFFDFF000 байт действительно добавляется к заданному адресу памяти.

Пример 4.5. Отображение управляющей области процессора (KPCR)

```
E:\>w2k_mem +df #0x54 0
[...]
FFDFF000..FFDFF053: 84 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
FFDFF000 | BECD9CF0 - BECD9DF0 : BECD6000 - 00000000 | sH?p sH•p sH`. ....
FFDFF010 | 00000000 - 00000000 : 7FFDE000 - FFDFF000 | .... .... .эа. яЯр.
FFDFF020 | FFDFF120 - 00000000 : 00000000 - 00000000 | яЯс  .... .... ....
FFDFF030 | FFFF20C0 - 00000000 : 80036400 - 80036000 | яя А .... ?.d. ?.`.
FFDFF040 | 80244000 - 00010001 : 00000001 - 000000C9 | ?$@. .... .... ...E
FFDFF050 | 00000000 -          :                     | ....
[...]
```

Адресация FS:[<base>]

При исследовании кода ядра Windows 2000 вы часто будете встречать такие инструкции, как MOV EAX и FS:[18h]. Эти инструкции извлекают значения членов структур TEB, KPCR или других структур, содержащихся в сегменте FS. Многие члены представляют собой указатели на другие внутренние структуры. Параметры командной строки +u и +k позволяют вам легко отслеживать такую косвенную адресацию. Параметр +u извлекает указатель из сегмента FS в режиме пользователя, параметр +k — делает то же в режиме ядра. Например, команда w2k_mem.exe +du #0x1E8 0x30 (пример 4.6) копирует 448 байт блока памяти, адресованного значением FS:[30h] в пользовательском режиме, который оказался указателем на блок переменных окружения процесса (PEB) w2k_mem.exe. Команда w2k_mem.exe +dk #0x1C 0x20 (пример 4.7) отображает первые 128 байт памяти, адресованные значением FS:[20h] в режиме ядра, что является указателем на управляющий блок процессора (Kernel's Processor Control Block, KPCB), кратко упомянутый ранее при обсуждении функции IOCTL SPY_IO_OS_INFO, он также будет обсуждаться далее в главе 7 (листинг 7.15). Пусть вас не беспокоит, что вы не знаете, что такое PEB или KPRCB, — вы будете это знать после прочтения этой книги.

Пример 4.6. Отображение блока переменных окружения процесса (PEB)

```
E:\>w2k_mem +du #0x1E8 0x30
[   ]
7FFDF000..7FFDF1E7: 488 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
7FFDF000 | 00000000 - FFFFFFFF : 00400000 - 00131E90 | .... яяяя .@.. ...•
7FFDF010 | 00020000 - 00000000 : 00130000 - 77FCD170 | .... .... .... wьCp
```

продолжение ↓

Пример 4.6 (*продолжение*)

```
7FFDF020 | 77F8AA4C - 77F8AA7D : 00000001 - 77E33E58 | wшCL wшC} .... wr>X
7FFDF030 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF040 | 77FCD1A8 - 00000001 : 00000000 - 7F6F0000 | wьCЁ .... .... .o..
7FFDF050 | 7F6F0000 - 7F6F0688 : 7FFB0000 - 7FFC1000 | .o.. .o.? .ы.. .ь..
7FFDF060 | 7FFD2000 - 00000001 : 00000000 - 00000000 | .э . .... .... ....
7FFDF070 | 079B8000 - FFFFE86D : 00100000 - 00002000 | .??. яяим .... .. .
7FFDF080 | 00010000 - 00001000 : 00000002 - 00000010 | .... .... .... ....
7FFDF090 | 77FCE380 - 00410000 : 00000000 - 00000014 | wьг? .A.. .... ....
7FFDF0A0 | 77FCD348 - 00000005 : 00000000 - 00000893 | wьУH .... .... ...?
7FFDF0B0 | 00000002 - 00000003 : 00000004 - 00000000 | .... .... .... ....
7FFDF0C0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0D0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0E0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF0F0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF100 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF110 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF120 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF130 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF140 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF150 | 77FCDCC0 - 00000000 : 00000000 - 00000000 | wьЬA .... .... ....
7FFDF160 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF170 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF180 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF190 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1A0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1B0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1C0 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
7FFDF1D0 | 00000000 - 00000000 : 00000000 - 00020000 | .... .... .... ....
7FFDF1E0 | 7F6F06C2 - 00000000 :           -         | .o.B ....
[   ]
```

Пример 4.7. Отображение управляющего блока процессора в ядре (KPRCB)

```
E:\>w2k_mem +dk #0x1C 0x20
[   ]
FFDFF120..FFDFF13B: 28 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
FFDFF120 | 00010001 - 86BBA820 : 00000000 - 8046BDF0 | .... ?»?  .... ?F??
FFDFF130 | 00020000 - 00000001 : 05010106 -          | .... .... ....
[   ]
```

Определение объекта по его дескриптору

Предположим, у вас есть дескриптор HANDLE объекта и вы хотите посмотреть, как выглядит в памяти соответствующий ему объект. Это задача почти тривиальна, если воспользоваться параметром +h, который вызывает функцию драйвера слежения SPY_IO_HANDLE_INFO (листинг 4.26) для поиска тела объекта по заданному дескриптору. Мир объектов Windows 2000 — чрезвычайно интересная тема, которая будет подробно рассмотрена в главе 7. Пока давайте на время отложим ее рассмотрение.

Относительная адресация

Иногда может понадобиться вывести ряд блоков памяти, размещенных через промежутки в одинаковое число байт. Это может быть, например, массив структур, такой как стек блоков TEB в многопоточном приложении. Такая относительная адресация включается параметрами +a и +s, которые изменяют значение указанного адреса на смещение. Параметры различаются тем, что +a («add bias», добавить смещение) задает положительное смещение, а +s («substract bias», вычесть смещение) — отрицательное. Пример 4.8 демонстрирует результат выполнения команды w2k_mem +d #32 0xC0000000 +a 4096 4096 в моей системе. Выводятся первые 32 байта трех расположенных друг за другом страниц размером 4 Кбайт, начиная с адреса 0xC0000000, по которому размещены системные таблицы страниц. Обратите внимание на переключатель +a в конце команды. Он определяет, что следующие лексемы «4096» следует трактовать как смещения, добавляемые к заданному раньше базовому адресу. Действие ключей +a и +s продолжается до тех пор, пока они не будут явным образом отключены заданием параметра –a или –s или не будет выполнена команда с другими ключами, изменяющими трактовку адреса.

Пример 4.8. Выборка таблиц страниц

```
E:\>w2k_mem +d #32 0xC0000000 +a 4096 4096
[...]
C0000000..C000001F: 32 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
C0000000 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....
C0000010 | 00000000 - 00000000 : 00000000 - 00000000 | .... .... .... ....

C0001000..C000101F: 32 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
C0001000 | 037D1025 - 03324025 : 0328D025 - 02F53025 | .}.% .2@% .)P% .Sa%
C0001010 | 06F17067 - 03297225 : 05115067 - 00000000 | .cpg .)r% ..Pg ....

C0002000..C000201F: 0 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
C0002000 |           -         :           -         |
C0002010 |           -         :           -         |

          96 bytes requested
          64 bytes received
```

В примере 4.8 также показано, что происходит при задании недопустимого линейного адреса. Первые две таблицы страниц для диапазонов адресов размером 4 Мбайт 0x00000000–0x003F0000 и 0x00400000–0x007F000 были допустимыми, а вот третья — нет, поэтому для нее утилита w2k_mem.exe показала пустую таблицу. Информация о допустимых адресах передается программе через струкутру SPY_MEMORY_ DATA (см. листинг 4.25), которую заполняет функция драйвера слежения SPY_IO_MEMORY_DATA.

Косвенная адресация

Одним из моих любимых параметров является +p, потому что он сэкономил мне немало времени при наборе во время подготовки этой книги. Параметр работает аналогично +u и +k, но в качестве сегмента для смещения использует не сегмент FS, а предыдущий отображенный блок. Это очень удобно, когда нужно пройти по связному списку объектов. Без этой возможности пришлось бы отобразить первый элемент списка, прочитать адрес следующего элемента, набрать команду с этим адресом и т. д. Теперь достаточно добавить в команду ключ +p и ряд смещений, показывающих, в каком месте предыдущего отображенного блока памяти находится ссылка на следующий объект.

В примере 4.9 я воспользовался этим ключом, чтобы пройти по списку активных процессов. Сначала я при помощи Kernel Debugger получил адрес внутренней переменной PsActiveProcessHead, структуры LIST_ENTRY, задающей начало списка процессов. Структура LIST_ENTRY состоит из члена Flink (forward link, связь с последующим) по смещению 0 и члена Blink (backward link, связь с предыдущим) по смещению 4 (см. листинг 2.7). Команда w2k_mem #8 +d 0x8046a180 +p 0 0 0 0 сначала считывает память, занятую структурой LIST_ENTRY переменной PsActiveProcessHead, и затем переходит на косвенную адресацию в соответствии с параметром +p. Четыре нуля показывают, что w2k_mem.exe нужно извлечь значение из предыдущего блока данных со смещением ноль, которое, конечно, представляет член Flink структуры LIST_ENTRY. Заметьте, что член Blink в примере 4.9, размещенный по смещению 4, и в самом деле указывает назад на предыдущую структуру LIST_ENTRY.

Если добавить в команду достаточное количество нулевых параметров, шестнадцатеричный снимок памяти в конце концов вернется к переменной PsActiveProcessHead, отмечающей начало и конец списка процессов. Как отмечалось в главе 2, поддерживаемые Windows 2000 дважды связные списки, как правило, цикличны, то есть Flink последнего элемента списка указывает на первый элемент, а Blink первого элемента — на последний.

Пример 4.9. Проход по списку активных процессов

```
E:\>w2k_mem #8 +d 0x8046a180 +p 0 0 0 0
[...]
B046A180..8046A187: 8 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|--------------------:--------------------|--------------------
8046A180 | 8149D900 - 840D2BE0 :          -          | •IЩ. ?.+a

8149D900 - B1490907: 8 valid bytes

Address   | 00000000 - 00000004 : 0000000B - 0000000C | 0000 0004 0008 000C
----------|--------------------:--------------------|--------------------
8149D900 | 8131A4A0 - 8046A1B0 :          -          | •1¤  ?FЎ?

8131A4A0 - 8131A4A7: B valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|--------------------:--------------------|--------------------
8131A4A0 | 812FFDE0 - 8149D900 :          -          | •/эа  •IЩ.
```

```
812FFDE0 - 812FFDE7: 8 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
812FFDE0  | 812FA460 - 8131A4A0 :           -         | •/¤` •l¤

812FA460 - 812FA467: 8 valid bytes

Address   | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
----------|---------------------:---------------------|--------------------
812FA460  | 812E30C0 - 812FFDE0 :           -         | •.0A •/ya
[...]
```

Загрузка модулей на лету

Иногда может понадобиться скопировать образ памяти модуля, который не отображен в линейное адресное пространство процесса w2k_mem.exe. Проблему можно решить, явно загрузив модуль при помощи параметров команды /<path> и +x. Каждая предваряемая слэшем лексема команды рассматривается как путь к модулю, и w2k_mem.exe попытается загрузить этот модуль при помощи функции Win32 API LoadLibraryEx(). По умолчанию используется параметр DON'T_RESOLVE_DLL_REFERENCES, указывающий, что модуль будет загружен без инициализации. Для библиотеки DLL это означает, что не будет вызываться ее точка входа DllMain(). Кроме того, не загружается ни один из зависимых модулей, указанных в секции импортируемых модулей. Если перед именем пути задать параметр +x, модуль будет загружен и полностью инициализирован. Заметьте, что некоторые модули не могут быть инициализированы в контексте процесса w2k_mem.exe. Например, драйверы устройств режима ядра не следует загружать с включенным параметром +x.

Как правило, загрузка и отображение модуля происходят в два этапа, как показано в примере 4.10. Сначала следует загрузить модуль без отображения каких-либо данных, чтобы найти назначенный ему системой базовый адрес. К счастью, адреса загрузки будут одинаковы, если к процессу за это время не были добавлены другие модули, поэтому при следующей попытке загрузить модуль базовый адрес будет таким же. В примере 4.10 я загрузил драйвер устройства режима ядра nwrdr.sys, представляющий собой редиректор NetWare в сетях Microsoft. На своем компьютере я не использую IPX\SPX, поэтому этот драйвер еще не загружен. Вызов функции API LoadLibraryEx() завершился успешно, и снимки памяти по возвращенному ей адресу 0x007A0000 до и после вызова свидетельствуют, что сначала эта область не использовалась, а затем в ней появился заголовок DOS.

Пример 4.10. Загрузка модуля и отображение его образа

```
E:\>w2k_mem  /e:\winnt\system32\drivers\nwrdr.sys
[...]
You didn't request any data!

LoadLibrary (e:\winnt\system32\drivers\nwrdr.sys) = 0x007A0000
[...]
E:\>w2k_mem 0x007A0000 /e:\winnt\system32\drivers\nwrdr.sys 0x007A0000
[...]
007A0000..007A00FF: 0 valid bytes
```

продолжение

Пример 4.10 (*продолжение*)

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|----------------
007A0000 |                         :                         |
007A0010 |                         :                         |
007A0020 |                         :                         |
007A0030 |                         :                         |
007A0040 |                         :                         |
007A0050 |                         :                         |
007A0060 |                         :                         |
007A0070 |                         :                         |
007A0080 |                         :                         |
007A0090 |                         :                         |
007A00A0 |                         :                         |
007A00B0 |                         :                         |
007A00C0 |                         :                         |
007A00D0 |                         :                         |
007A00E0 |                         :                         |
007A00F0 |                         :                         |

LoadLibrary (e:\winnt\system32\drivers\nwrdr.sys) = 0x007A0000

007A0000..007A00FF: 256 valid bytes

Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|----------------
007A0000 | 4D 5A 90 00-03 00 00 00 : 04 00 00 00-FF FF 00 00 | MZ•.........яя..
007A0010 | B8 00 00 00-00 00 00 00 : 40 00 00 00-00 00 00 00 | ё.......@.......
007A0020 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
007A0030 | 00 00 00 00-00 00 00 00 : 00 00 00 00-D0 00 00 00 | ............Р...
007A0040 | 0E 1F BA 0E-00 B4 09 CD : 21 B8 01 4C-CD 21 54 68 | ..є..r.H!ё.LH!Th
007A0050 | 69 73 20 70-72 6F 67 72 : 61 6D 20 63-61 6E 6E 6F | is program canno
007A0060 | 74 20 62 65-20 72 75 6E : 20 69 6E 20-44 4F 53 20 | t be run in DOS
007A0070 | 6D 6F 64 65-2E 0D 0D 0A : 24 00 00 00-00 00 00 00 | mode....$.......
007A0080 | 61 14 4B C1-25 75 25 92 : 25 75 25 92-25 75 25 92 | a.KA%u%?%u%?%u%?
007A0090 | 29 55 2B 92-27 75 25 92 : 7C 56 36 92-22 75 25 92 | )U+?'u%?|V6?"u%?
007A00A0 | 25 75 24 92-BF 75 25 92 : 0F 7D 23 92-24 75 25 92 | %u$?u%?.}#?$u%?
007A00B0 | 25 75 25 92-14 75 25 92 : 52 69 63 68-25 75 25 92 | %u%?.u%?Rich%u%?
007A00C0 | 00 00 00 00-00 00 00 00 : 00 00 00 00-00 00 00 00 | ................
007A00D0 | 50 45 00 00-4C 01 09 00 : 66 EC 0B 38-00 00 00 00 | PE..L...fм.8....
007A00E0 | 00 00 00 00-E0 00 0E 03 : 0B 01 05 0C-00 2D 02 00 | ....а........-..
007A00F0 | 40 3A 00 00-00 00 00 00 : 3E 14 01 00-40 03 00 00 | @:......>...@...
[...]
```

Удивительно, но при помощи параметра /<path> можно загрузить в память даже файл .exe другого приложения. Однако этот модуль, скорее всего, будет загружен по нестандартному адресу, поскольку его типичный адрес загрузки обычно уже занят w2k_mem.exe. Более того, загруженное приложение не удастся выполнить — переключатель +x применим только к библиотекам DLL и не работает с другими типами модулей.

Изменение памяти при подкачке страниц по запросу

При обсуждении функции драйвера слежения SPY_IO_MEMORY_DATA я упомянул о том, что при помощи этой функции можно читать содержимое страниц памяти, выгруженных в файл подкачки. Пришло время проверить это утверждение. Во-пер-

вых, необходимо привести систему в состояние острой нехватки памяти, чтобы она выгрузила в файл подкачки все, в чем нет явной необходимости. Мой любимый способ достичь этого состоит в следующем:

1. Скопируйте экран рабочего стола Windows 2000 в буфер обмена, нажав на клавишу PrintScreen.
2. Вставьте это растровое изображение в приложение для работы с графикой.
3. Раздвиньте рисунок до очень больших размеров.

Теперь давайте посмотрим, что выведет на экран команда w2k_mem +d #16 0xC0280000 0xA0000000 0xA0001000 0xA0002000 0xC0280000. Что это за команда? Она просто делает снимок памяти некоторых записей PTE до и после обращения к страницам, на которые они ссылаются. Четыре записи PTE, расположенные по адресу 0xC0280000, принадлежат диапазону линейных адресов 0xA0000000–0xA003FFF, который является частью образа модуля ядра win32k.sys. Как видно из примера 4.11, этот диапазон адресов был выгружен из физической памяти благодаря предыдущим операциям с растровым рисунком. Откуда это известно? Об этом свидетельствует тот факт, что четыре 32-битных значения по адресу 0xC0280000 — четные числа, то есть их самый младший бит, бит P в записи PTE, равен нулю, а это означает, что страницы нет в физической памяти. Следующих три снимка памяти соответствуют параметрам команды 0xA0000000, 0xA0001000 и 0xA0002000, которые запрашивают данные о трех исследуемых страницах, не затрагивая четвертую. Как видно, w2k_mem.exe без труда смог обратиться к этим страницам — система просто загрузила их в физическую память по запросу. Остался еще один тест: как выглядят четыре записи PTE после выполненных команд? Ответ даст последний снимок памяти в примере 4.11: у первых трех записей PTE бит P установлен, а у четвертой записи — нет, свидетельствуя о ее отсутствии в физической памяти.

Пример 4.11. Изменение состояния таблиц PTE

```
E:\>w2k_mem +d #16 0xC0280000 0xA0000000 0xA0001000 0xA000200 0xC0280000
[...]
C0280000..C028000F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C02B0000 | 056A14E0 - 056A14E2 : 056A14E4 - 056A14E6 | .j.а .j.в .j.д .j.ж

A0000000..A000000F: 16 valid bytes

Address  | 00000000 - 00000004 : 0000000B - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
A0000000 | 00905A4D - 00000003 : 00000004 - 0000FFFF | .•ZM .... .... ..яя

A0001000..A000100F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 000B 000C
---------|---------------------:---------------------|--------------------
A0001000 | 000000A6 - FF0C75FF : 1738B415 - FB458BA0 | ...| я.uя .8г. шЕ?
```

продолжение ⇩

Пример 4.11 (*продолжение*)

```
A0002000..A000200F: 16 valid bytes

Address  | 00000000 - 00000004 : 0000000B - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
A0002000 | 89A018E0 - F685D875 : 468D1A74 - 458D5020 | ? .a u?Ыu F•.t E•P

C0280000..C028000F: 16 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|--------------------
C0280000 | 0556B123 - 028C2121 : 05AD1121 - 056A14E6 | .V±# .?!! .-.! .j.?
```

Перед тем как перейти к следующему разделу, пожалуйста, еще раз взгляните на первый шестнадцатеричный снимок в примере 4.11. Первые четыре записи PTE по адресу 0xC0280000 весьма похожи друг на друга. На самом деле они отличаются только тремя младшими битами. Если вы изучите еще несколько таких записей PNPE, ссылающихся на страницы в файлах подкачки, вы убедитесь, что у всех у них установлен бит № 10. Вот почему в листинге 4.3 я назвал этот бит PageFile. Если он установлен, остальные биты, за исключением флага P, вероятно, указывают положение этой страницы в файлах подкачки.

Дополнительные параметры командной строки

Некоторые из самых интересных параметров команды, перечисленных в примере 4.1, еще не обсуждались. Например, мы не говорили о «параметрах состояния системы» +o, +c, +g, +i и +b, хотя их общее название звучит многообещающе. Я еще вернусь к ним в последнем разделе этой главы, в котором будет раскрыт ряд секретов памяти Windows 2000.

Взаимодействие с драйвером слежения

Теперь, когда вы знаете, как пользоваться утилитой w2k_mem.exe, можно перейти к рассмотрению деталей ее работы. Вместо обсуждения синтаксиса командной строки давайте лучше посмотрим, как это приложение взаимодействует с драйвером слежения из модуля w2k_spy.sys.

Еще раз о механизме управления вводом-выводом

Принцип работы механизма IOCTL со стороны ядра уже был показан в листингах 4.6 и 4.7. Драйвер слежения просто ожидает пакеты запроса ввода-вывода (I/O Request Packets, IRP) и обрабатывает некоторые из них, в основном запросы, помеченные как IRP_MJ_DEVICE_CONTROL, требующие выполнения определенных запрещенных действий, запрещенных по крайней мере в контексте отправившего запрос приложения пользовательского режима. Это реализовано посредством вызова функции Win32 API DeviceIoControl(), прототип которой приведен в листинге 4.27. Аргументы dwIoControlCode, lpInBuffer, nInBufferSize, lpOutBuffer, nOutBufferSize и lpBytesReturned должны показаться вам знакомыми. Фактически они один к одно-

му соответствуют аргументам dcode, pInput, dInput, pOutput, dOutput и pdInfo функции SpyDispatcher() в листинге 4.7. Вот краткое описание остальных аргументов: hDevice — дескриптор устройства драйвера слежения, lpOverlapped может указывать на структуру OVERLAPPED, требуемую для асинхронного механизма IOCTL. Мы не собираемся отправлять асинхронные запросы, поэтому этот аргумент всегда будет равен NULL.

В листинге 4.28 представлен набор функций-оболочек, выполняющих основные операции IOCTL. Самая важная функция — IoControl(), которая вызывает DeviceIoControl() и проверяет результат ее работы — размер выходных данных. Поскольку w2k_mem.exe точно задает размеры своих выходных буферов, число выходных байт должно всегда быть равно размеру буфера. ReadBinary() — это упрощенная версия IoControl() для функций IOCTL, не требующих входных данных. Функции ReadCpuInfo(), ReadSegment() и ReadPhysical() специально разработаны для наиболее часто используемых функций драйвера слежения SPY_IO_CPU_INFO, SPY_IO_SEGMENT и SPY_IO_PHYSICAL. Представление их в виде функций C значительно повышает удобочитаемость кода.

Листинг 4.27. Прототип функции DeviceIoControl()

```
BOOL WINAPI DeviceIoControl (HANDLE        hDevice,
                             DWORD         dwIoControlCode,
                             PVOID         lpInBuffer,
                             DWORD         nInBufferSize,
                             PVOID         lpOutBuffer,
                             DWORD         nOutBufferSize,
                             PDWORD        lpBytesReturned,
                             PDVERLAPPED   lpOverlapped);
```

Листинг 4.28. Различные функции-оболочки IOCTL

```
BOOL WINAPI IoControl (HANDLE  hDevice,
                       DWORD   dCode,
                       PVDID   pInput,
                       DWORD   dInput,
                       PVDID   pOutput,
                       DWORD   dOutput)
    {
    DWORD dData = 0;

    return DeviceIoControl (hDevice,  dCode,
                            pInput,   dInput,
                            pOutput,  dOutput,
                            &dData,   NULL)
           &&
           (dData == dOutput);
    }

//  -------------------------------------------------------

BOOL WINAPI ReadBinary (HANDLE  hDevice,
                        DWORD   dCode,
                        PVOID   pOutput,
                        DWORD   dOutput)
```

продолжение

Листинг 4.28 (*продолжение*)

```
    {
    return IoControl (hDevice, dCode, NULL, 0,
                      pOutput, dOutput);
    }

//  ----------------------------------------------------

BOOL WINAPI ReadCpuInfo (HANDLE          hDevice,
                         PSPY_CPU_INFO   psci)
    {
    return IoControl (hDevice,  SPY_IO_CPU_INFO,
                      NULL,     0,
                      psci,     SPY_CPU_INFO_);
    }

//  ----------------------------------------------------

BOOL WINAPI ReadSegment (HANDLE          hDevice,
                         DWORD           dSelector,
                         PSPY_SEGMENT    pss)
    {
    return IoControl (hDevice,       SPY_IO_SEGMENT,
                      &dSelector,    DWORD_,
                      pss,           SPY_SEGMENT_);
    }

//  ----------------------------------------------------

BOOL WINAPI ReadPhysical (HANDLE              hDevice,
                          PVOID               pLinear,
                          PPHYSICAL_ADDRESS   ppa)
    {
    return IoControl (hDevice,    SPY_ID_PHYSICAL,
                      &pLinear,   PVOID_,
                      ppa,        PHYSICAL_ADDRESS_)
           &&
           (ppa->LowPart || ppa->HighPart);
    }
```

Всем показанным до сих пор функциям требовался дескриптор устройства драйвера слежения, и теперь я покажу, как получить его значение. В действительности это весьма простая операция Win32, похожая на открытие файла. В листинге 4.29 показана реализация обработчика команд внутри w2k_mem.exe. В этом коде используются функции API w2kFilePath(), w2kServiceLoad() и w2kServiceUnload(), экспортируемые библиотекой утилит «SBS Windows 2000 Utility Library» w2k_lib.dll, содержащейся на прилагающемся к книге компакт-диске. Если вы читали раздел о Service Control Manager Windows 2000 в главе 3, вам будут уже знакомы функции w2kServiceLoad() и w2kServiceUnload() из листинга 3.8. Эти мощные функции на лету загружают и выгружают драйверы устройств режима ядра и обрабатывают неопасные ошибки, такие как загрузка драйвера, который уже загружен. Функция w2kFilePath() — это вспомогательная функция, получающая путь к файлу из основного пути по имени или расширению файла. w2k_mem.exe вызывает ее, что-

бы получить полностью заданный путь к исполняемому файлу драйвера слежения, согласованный со своим собственным путем.

Листинг 4.29. Управление драйвером слежения

```
WORD awSpyFile          [] = SW(DRV_FILENAME);
WORD awSpyDevice        [] = SW(DRV_MODULE);
WORD awSpyDisplay       [] = SW(DRV_NAME);
WORD awSpyPath          [] = SW(DRV_PATH);

//  ------------------------------------------------------

void WINAPI Execute (PPWORD ppwArguments,
                     DWORD  dArguments)
     {
     SPY_VERSION_INFO   svi;
     DWORD              dOptions, dRequest, dReceive;
     WORD               awPath [MAX_PATH]  = L"?";
     SC_HANDLE          hControl           = NULL;
     HANDLE             hDevice            = NULL[1];

     _printf (L"\r\nLoading \"%s\" (%s) ...\r\n",
              awSpyDisplay, awSpyDevice);

     if (w2kFilePath (NULL, awSpyFile, awPath, MAX_PATH))
          {
          _printf (L"Driver: \"%s\"\r\n",
                   awPath);

          hControl = w2kServiceLoad (awSpyDevice,
                                     awSpyDisplay,
                                     awPath, TRUE);
          }
     if (hControl != NULL)
          {
          _printf (L"Opening \"%s\" ...\r\n",
                   awSpyPath);

          hDevice = CreateFile (awSpyPath, GENERIC_READ,
                                FILE_SHARE_READ | FILE_SHARE_WRITE,
                                NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, NULL);
          }
     if (hDevice != INVALID_HANDLE_VALUE)[2]
          {
          if (ReadBinary (hDevice, SPY_ID_VERSION_INFO,
```

[1] На компакт-диске другое значение: «INVALID_HANDLE_VALUE». — *Примеч. перев.*

[2] В коде на компакт-диске есть еще ветвь else:

```
else
  {
  _printf (L"Unable to load the spy device driver.\r\n");
  }
```

Примеч. перев.

Листинг 4.29 (*продолжение*)

```
                    &svi, SPY_VERSION_INFO_))
            {
            _printf (L"\r\n%s V%lu.%02lu ready\r\n",
                    svi.awName,
                    svi.dVersion / 100, svi.dVersion % 100);
            }
        dOptions = COMMAND_OPTION_NONE;
        dRequest = CommandParse (hDevice, ppwArguments,
                                 dArguments, TRUE, &dOptions);

        dOptions = COMMAND_OPTION_NONE;
        dReceive = CommandParse (hDevice, ppwArguments,
                                 dArguments, FALSE, &dOptions);
        if (dRequest)
            {
            _printf (awSummary,
            dRequest, (dRequest == 1 ? awByte : awBytes),
            dReceive, (dReceive == 1 ? awByte : awBytes));
            }
        _printf (L"\r\nClosing the spy device ...\r\n");
        CloseHandle (hDevice);
        }
    else
        {
        _printf (L"Unable to open the spy device.\r\n");
        }
    if ((hControl != NULL) && gfSpyUnload)
        {
        _printf (L"Unloading the spy device ...\r\n");
        w2kServiceUnload (awSpyDevice, hControl);
        }
    return;
    }
```

Обратите внимание на четыре определения глобальных переменных-строк в начале листинга 4.29. Константы DRV_FILENAME, DRV_MODULE, DRV_NAME и DRV_PATH взяты из заголовочного файла драйвера слежения w2k_spy.h. В табл. 4.4 приведены их текущие значения. В исходных файлах w2k_mem.exe нет никаких определений, относящихся к устройству. Все необходимое для клиентского приложения предоставляет файл w2k_spy.h. Это очень важно: если в будущем изменятся какие-либо связанные с устройством определения, не нужно будет обновлять файлы приложений. Достаточно будет только заново скомпоновать приложение с новым заголовочным файлом программы просмотра памяти.

Вызов w2kFilePath() в начале листинга 4.29 гарантирует, что файл w2k_spy.sys, определенный в глобальной строке awSpyFile (табл. 4.4), всегда будет загружаться из того каталога, где находится w2k_mem.exe. После этого код листинга 4.29 пытается загрузить и запустить драйвер слежения вызовом функции w2kServiceLoad(), передав ей в качестве параметров глобальные строки awSpyDevice и awSpyDisplay (табл. 4.4). Если драйвер еще не был загружен, эти строки будут записаны в список свойств драйвера, где другие приложения смогут получить их значения, иначе сохранятся текущие настройки свойств. Хотя функция w2kServiceLoad() в листинге 4.29

возвращает дескриптор, это не тот дескриптор, который можно использовать в вызовах IOCTL. Чтобы получить дескриптор устройства драйвера слежения, необходимо воспользоваться многоцелевой функцией Win32 CreateFile(). Эта функция открывает или создает практически все, что можно открыть или создать в Windows 2000. Вы несомненно много раз вызывали эту функцию, чтобы получить дескриптор файла. CreateFile() может открывать и устройства режима ядра, если имя символьной ссылки (symbolic link) устройства передано ей в формате \\.\<SymbolicLink> через аргумент lpFileName. Символьная ссылка устройства драйвера слежения называется w2k_spy, поэтому первый аргумент CreateFile() должен называться \\.\w2k_spy, это значение содержит глобальная переменная-строка awSpyPath (табл. 4.4).

Таблица 4.4. Определения описывающих устройство строк

Константа w2k_spy	Переменная w2k_mem	Значение
DRV_FILENAME	awSpyFile	w2k_spy.sys
DRV_MODULE	awSpyDevice	w2k_spy
DRV_NAME	awSpyDisplay	SBS Windows 2000 Spy Device
DRV_PATH	awSpyPath	\\.\w2k_spy

Если функция CreatrFile() завершится успешно, она вернет дескриптор устройства, который можно передать функции DeviceIoControl(). Функция Execute() из листинга 4.29 сразу же использует этот указатель для запроса информации о версии драйвера слежения, которую он отображает на экране в случае успешного завершения вызова IOCTL. После этого дважды вызывается функция CommandParse() с различным значением типа BOOL для четвертого аргумента. Первый вызов просто проверяет допустимость параметров в командной строке и показывает информацию об ошибках, фактическое выполнение всех команд происходит во втором вызове. Подробно обсуждать синтаксический анализатор команд я не хочу. Оставшийся код в листинге 4.29 освобождает ресурсы, закрывая дескрипторы и выгружая (необязательно) драйверы слежения. Некоторые другие фрагменты исходного кода w2k_mem.exe также представляют интерес, но здесь я не буду их обсуждать. Все детали можно узнать из файлов w2k_mem.c и w2k_mem.h в каталоге \src\w2k_mem компакт-диска с примерами.

Осталось обсудить еще одну достойную внимания деталь: проверку флага gfSpyUnload перед выгрузкой драйвера. Этот глобальный флаг установлен в FALSE, поэтому драйвер не будет выгружаться автоматически. Так сделано в целях повышения производительности w2k_mem.exe и других клиентов w2k_spy.sys, поскольку загрузка драйвера занимает определенное время. Загрузка потребует времени для первого клиента, но все последующие смогут воспользоваться наличием драйвера в памяти. При этом исключается возможность конфликтных ситуаций при одновременной работе клиентов, когда один клиент пытается выгрузить драйвер, в то время как другой его еще использует. Конечно, Windows 2000 не будет выгружать драйвер, пока не будут закрыты все дескрипторы его устройств, однако она переведет его в состояние STOP_PENDING, которое не позволит новым клиентам обращаться к устройству. Несмотря на это, если вы не будете запускать w2k_sys в многопользовательской среде и не собираетесь часто обновлять драйвер устройства, флаг gfSpyUnload можно установить в TRUE.

Внутренняя организация памяти Windows 2000

Помимо глобального разделения 4 Гбайт адресного пространства на части пользовательского режима и режима ядра эти две половины пространства делятся далее на многочисленные блоки меньшего размера. Как можно догадаться, в большинстве таких блоков хранятся недокументированные структуры, предназначенные для недокументированных целей. Проще всего было бы просто забыть о них, если бы они не представляли никакого интереса. Однако это не так — некоторые участки памяти являются просто золотым дном для разработчиков средств диагностики системы или отладочного программного обеспечения.

Основная информация об операционной системе

Теперь можно объяснить один из пропущенных параметров командной строки приложения для просмотра памяти w2k_mem.exe. Если взглянуть на нижнюю половину справочного экрана программы в примере 4.1, можно увидеть серию параметров под названием «System Status Options» (параметры состояния системы). Рассмотрим параметр +o по имени «display OS information» (показать информацию о системе), образец выполнения команды с этим параметром на моем компьютере приведен в примере 4.12. Отображенные данные представляют собой содержимое структуры SPY_OS_IO, определенной в листинге 4.13, значения полей которой задает функция драйвера слежения SpyOutputInfo() из того же листинга. В примере 4.12 также отображены некоторые особенные адреса в 4 Гбайт линейного адресного пространства процесса. Например, диапазон допустимых адресов пользовательского режима выведен как 0x00010000–0x7FFEFFFF. Вам, возможно, известно из других книг по программированию для Windows NT или Windows 2000, что первые и последние 64 Кбайт отведенной пользовательскому режиму половины линейной памяти — это «закрытые разделы», доступ к которым закрыт для обработки распространенных ошибок программирования, связанных с неверными значениями указателей. Результат работы w2k_mem.exe подтверждает это.

Пример 4.12. Отображение информации об операционной системе

```
E:\>w2k_mem +o
[...]
SBS Windows 2000 Spy Device V1.00 ready

OS information:
---------------

Memory page size              : 4096 bytes
Memory page shift             : 12 bits
Memory PTI  shift             : 12 bits
Memory PDI  shift             : 22 bits
Memory page mask              : 0xFFFFF000
Memory PTI  mask              : 0x003FF000
Memory PDI  mask              : 0xFFC00000
Memory PTE  array             : 0xC0000000
Memory PDE  array             : 0xC0300000
```

```
Lowest user address            : 0x00010000
Thread environment block       : 0x7FFDE000
Highest user address           : 0x7FFEFFFF
User probe address             : 0x7FFF0000
System range start             : 0x80000000
Lowest system address          : 0xC0800000
Shared user data               : 0xFFDF0000
Processor control region       : 0xFFDFF000
Processor control block        : 0xFFDFF120

Global flag                    : 0x00000000
i386 machine type              : 0
Number of processors           : 1
Product type                   : Windows NT Workstation (1)
Version & Build number         : 5.00.2195
System root                    : "H:\WINNT"
[...]
```

В последних трех строках примера 4.12 приводится содержательная информация о системе, главным образом полученная из области SharedUserData по адресу 0xFFDF0000. Система записывает в эту область структуру данных KUSER_SHARED_DATA, которая определена в заголовочном файле DDK ntddk.h.

Сегменты и дескрипторы Windows 2000

Еще один очень удобный параметр w2k_mem.exe, +c отображает и распознает содержимое сегментных регистров процессора и таблицы дескрипторов. В примере 4.13 показан типичный результат работы команды. Содержимое сегментных регистров CS, DS и ES ясно показывает, что Windows 2000 предоставляет каждому процессу 4 Гбайт адресного пространства: значения базовых сегментов начинаются с 0x00000000 и ограничены 0xFFFFFFFF.

Символы флага в самом правом столбце показывают тип сегмента, как он определен в члене структуры Type дескриптора. Атрибуты типа сегментов данных и кода образованы комбинациями символов «cra» и «ewa» соответственно. Тире означает, что соответствующий атрибут не установлен. Для сегмента состояния задачи (Task State Segment, TSS) возможны только атрибуты «a» (available, свободен) и «b» (busy, занят). Список всех возможных атрибутов приведен в табл. 4.5. В примере 4.13 показано, что сегменты CS в Windows 2000 — несогласующиеся (nonconforming) и разрешают доступ на выполнение и чтение, в то время как сегменты DS, ES, FS и SS — расширяющегося типа и разрешают доступ на чтение и на запись. Еще одна не бросающаяся в глаза, но важная деталь — различие уровней привилегий дескриптора (Desscriptor Privilege Level, DPL) сегментов CS, FS и SS в режиме пользователя и режиме ядра. Для несогласующихся сегментов кода DPL задает уровень привилегий, в котором должен выполняться вызывающий код, чтобы ему было разрешено обращаться в этот сегмент (Intel, 1999c, pp. 4–8f). В пользовательском режиме требуется третий уровень, в режиме ядра — нулевой уровень. Для сегментов данных DPL определяет минимальный уровень привилегий, необходимый для доступа к сегменту. Это означает, что сегменты FS и SS в пользовательском режиме доступны из всех уровней привилегий, в режиме ядра разрешены обращения только из уровня 0.

Пример 4.13. Отображение информации процессора

```
E:\>w2k_mem +c
[...]
CPU information:
----------------

User mode segments:

CS    : Selector = 001B, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = CODE -ra
DS    : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
ES    : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
FS    : Selector = 0038, Base = 7FFDE000, Limit = 00000FFF, DPL3, Type = DATA -wa
SS    : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
TSS   : Selector = 0028, Base = 80244000, Limit = 000020AB, DPL0, Type = TSS32 b

Kernel mode segments:

CS    : Selector = 0008, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = CODE -ra
DS    : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
ES    : Selector = 0023, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
FS    : Selector = 0030, Base = FFDFF000, Limit = 00001FFF, DPL0, Type = DATA -wa
SS    : Selector = 0010, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = DATA -wa
TSS   : Selector = 0028, Base = 80244000, Limit = 000020AB, DPL0, Type = TSS32 b

IDT   : Limit    = 07FF, Base = 80036400
GDT   : Limit    = 03FF, Base = 80036000
LDT   : Selector = 0000

CR0   : Contents = 8001003B
CR2   : Contents = 7FFD3012
CR3   : Contents = 007E0000
[...]
```

Содержимое регистров IDT и GDT показывает, что таблица GDT расположена с линейного адреса 0x80036000 по адрес 0x800363FF, а сразу за ней находится таблица IDT, занимающая диапазон адресов 0x80036400–80036BFF. Каждый дескриптор занимает 64 бита, и в таблице GDT содержится 128 записей, в таблице IDT — 256. Заметьте, что GDT может содержать до 8192 записей, но Windows 2000 использует из них только небольшую часть.

У программы w2k_mem.exe есть еще два параметра командной строки, +g и +i, отображающих дополнительную информацию о таблицах GDT и IDT. В примере 4.14 показан результат выполнения команды с параметром +g. Они аналогичны разделу сегментов режима ядра в примере 4.13, но выводят значения всех доступных в режиме ядра селекторов сегментов, а не только селекторов сегментных регистров. Утилита w2k_mem.exe получает этот список, проходя в цикле по всей таблице GDT, запрашивая у драйвера слежения информацию о сегменте средствами функции IOCTL SPY_IO_SEGMENT. Отображаются только допустимые селекторы. Поучительно сравнить примеры 4.13 и 4.14 с определениями селекторов GDT в файле ntddk.h, собранными в табл. 4.6. Видно, что они соответствуют информации, полученной при помощи w2k_mem.exe.

Таблица 4.5. Атрибуты типа сегментов кода и данных

Сегмент	Атрибут	Описание
CODE	c	Согласующийся сегмент (возможны обращения из кода с меньшим уровнем привилегий)
CODE	r	Разрешен доступ на чтение (в отличие от доступа только на выполнение)
CODE	a	К сегменту было обращение
DATA	e	Сегмент растет вниз (типичный атрибут для сегментов стека)
DATA	w	Разрешен доступ на запись (в отличие от доступа только на чтение)
DATA	a	К сегменту было обращение
TSS32	a	Сегмент состояния задачи (TSS) свободен
TSS32	b	Сегмент состояния задачи занят

Пример 4.14. Отображение дескрипторов GDT

```
E:\>w2k_mem +g
[...]
GDT information:
----------------

001 : Selector = 0008, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = CODE -ra
002 : Selector = 0010, Base = 00000000, Limit = FFFFFFFF, DPL0, Type = DATA -wa
003 : Selector = 0018, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = CODE -ra
004 : Selector = 0020, Base = 00000000, Limit = FFFFFFFF, DPL3, Type = DATA -wa
005 : Selector = 0028, Base = 80244000, Limit = 000020AB, DPL0, Type = TSS32 b
006 : Selector = 0030, Base = FFDFF000, Limit = 00001FFF, DPL0, Type = DATA -wa
007 : Selector = 0038, Base = 7FFDE000, Limit = 00000FFF, DPL3, Type = DATA -wa
008 : Selector = 0040, Base = 00000400, Limit = 0000FFFF, DPL3, Type = DATA -wa
009 : Selector = 0048, Base = E2E6A000, Limit = 00000177, DPL0, Type = LDT
00A : Selector = 0050, Base = 80470040, Limit = 00000068, DPL0, Type = TSS32 a
00B : Selector = 0058, Base = 804700A8, Limit = 00000068, DPL0, Type = TSS32 a
00C : Selector = 0060, Base = 00022AB0, Limit = 0000FFFF, DPL0, Type = DATA -wa
00D : Selector = 0068, Base = 000B8000, Limit = 00003FFF, DPL0, Type = DATA -w-
00E : Selector = 0070, Base = FFFF7000, Limit = 000003FF, DPL0, Type = DATA -w-
00F : Selector = 0078, Base = 80400000, Limit = 0000FFFF, DPL0, Type = CODE -r-
010 : Selector = 0080, Base = 80400000, Limit = 0000FFFF, DPL0, Type = DATA -w-
011 : Selector = 0088, Base = 00000000, Limit = 00000000, DPL0, Type = DATA -w-
014 : Selector = 00A0, Base = 814985A8, Limit = 00000068, DPL0, Type = TSS32 a
01C : Selector = 00E0, Base = F0430000, Limit = 0000FFFF, DPL0, Type = CODE cra
01D : Selector = 00E8, Base = 00000000, Limit = 0000FFFF, DPL0, Type = DATA -w-
01E : Selector = 00F0, Base = 8042DCE8, Limit = 000003B7, DPL0, Type = CODE ---
01F : Selector = 00F8, Base = 00000000, Limit = 0000FFFF, DPL0, Type = DATA -w-
020 : Selector = 0100, Base = F0440000, Limit = 0000FFFF, DPL0, Type = DATA -wa
021 : Selector = 0108, Base = F0440000, Limit = 0000FFFF, DPL0, Type = DATA -wa
022 : Selector = 0110, Base = F0440000, Limit = 0000FFFF, DPL0, Type = DATA -wa
[...]
```

Таблица 4.6. Селекторы GDT, определенные в ntddk.h

Идентификатор	Значение	Комментарии
KGDT_NULL	0x0000	Селектор нулевого сегмента (недопустим
KGDT_R0_CODE	0x0008	Ргистр CS в режиме ядра
KGDT_R0_DATA	0x0010	Регистр SS в режиме ядра

продолжение

Таблица 4.6 (*продолжение*)

Идентификатор	Значение	Комментарии
KGDT_R3_CODE	0x0018	Ргистр CS в пользовательском режиме
KGDT_R3_DATA	0x0020	Регистры DS, ES и SS в пользовательском режиме, регистры DS и ES в режиме ядра
KGDT_TSS	0x0028	Сегмент состояния задачи (TSS) в режимах пользователя и ядра
KGDT_R0_PCR	0x0030	Регистр FS в режиме ядра (управляющая область процессора, Processor Control Region)
KGDT_R3_TEB	0x0038	Регистр FS в пользовательском режиме (блок переменных окружения потока, Thread Environment Block)
KGDT_VDM_TILE	0x0040	База 0x00000400, граница 0x0000FFFF (виртуальная машина DOS)
KGDT_LDT	0x0048	Локальная таблица дескрипторов (LDT)
KGDT_DF_TSS	0x0050	Переменная KiDoubleFaultTSS модуля ntoskrnl.exe
KGDT_NML_TSS	0x0058	Переменная KiNMITSS модуля ntoskrnl.exe

Назначение некоторых селекторов из примера 4.14, отсутствующих в табл. 4.6, можно определить, если осуществить поиск похожих базовых адресов или содержимого памяти и затем сопоставить идентификаторы для базовых адресов при помощи Kernel Debugger. В табл. 4.7 приведены определенные мною таким образом селекторы.

Параметр +i программы w2k_mem.exe копирует из памяти значения дескрипторов шлюзов из таблицы IDT. В примере 4.15 приведена часть этого весьма длинного списка, — первые 20 записей, значение которых предопределено и назначается Intel (Intel, 1999c, pp. 5–6). Прерывания с 0x14 по 0x1F зарезервированы Intel, остальной диапазон 0x20–0xFF доступен для операционной системы.

В табл. 4.8 я свел вместе все прерывания для опознаваемых и нетривиальных шлюзов прерываний, ловушек и задач. Как объяснялось раньше в этой главе, большинство определенных пользователем прерываний указывают на фиктивные обработчики с именем KiUnexpectedInterrupNNN(). Некоторые обработчики прерываний размещены по адресам, которым нельзя сопоставить идентификаторы при помощи Kernel Debugger.

Таблица 4.7. Другие селекторы GDT

Значение	База	Описание
0x0078	0x80400000	Сегмент кода ntoskrnl.exe
0x0080	0x80400000	Сегмент данных ntoskrnl.exe
0x00A0	0x814985A8	TSS (член EIP указывает на HalpMcaExceptionHandlerWrapper)
0x00E0	0xF0430000	Сегмент кода ROM BIOS
0x00F0	0x8042DCE8	Функция ntoskrnl.exe Ki386CallAbios
0x0100	0xF0440000	Сегмент данных ROM BIOS
0x0108	0xF0440000	Сегмент данных ROM BIOS
0x0110	0xF0440000	Сегмент данных ROM BIOS

Пример 4.15. Дескрипторы шлюзов из таблицы IDT

```
E:\>w2k_mem +i
[...]
IDT information:
----------------

00 : Pointer = 0008:804625E6, Base = 00000000, Limit = FFFFFFFF, Type = INT32
01 : Pointer = 0008:80462736, Base = 00000000, Limit = FFFFFFFF, Type = INT32
02 : TSS     = 0058,          Base = 804700A8, Limit = 00000068, Type = TASK
03 : Pointer = 0008:80462A0E, Base = 00000000, Limit = FFFFFFFF, Type = INT32
04 : Pointer = 0008:80462B72, Base = 00000000, Limit = FFFFFFFF, Type = INT32
05 : Pointer = 0008:80462CB6, Base = 00000000, Limit = FFFFFFFF, Type = INT32
06 : Pointer = 0008:80462E1A, Base = 00000000, Limit = FFFFFFFF, Type = INT32
07 : Pointer = 0008:80463350, Base = 00000000, Limit = FFFFFFFF, Type = INT32
08 : TSS     = 0050,          Base = 80470040, Limit = 00000068, Type = TASK
09 : Pointer = 0008:8046370C, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0A : Pointer = 0008:80463814, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0B : Pointer = 0008:80463940, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0C : Pointer = 0008:80463C44, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0D : Pointer = 0008:80463E50, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0E : Pointer = 0008:804648A4, Base = 00000000, Limit = FFFFFFFF, Type = INT32
0F : Pointer = 0008:80464C3F, Base = 00000000, Limit = FFFFFFFF, Type = INT32
10 : Pointer = 0008:80464D47, Base = 00000000, Limit = FFFFFFFF, Type = INT32
11 : Pointer = 0008:80464E6B, Base = 00000000, Limit = FFFFFFFF, Type = INT32
12 : TSS     = 00A0,          Base = 814985A8, Limit = 00000068, Type = TASK
13 : Pointer = 0008:80464C3F, Base = 00000000, Limit = FFFFFFFF, Type = INT32
[...]
```

Таблица 4.8. Шлюзы прерываний, ловушек и задач Windows 2000

INT	Описание INTEL	Владелец	Обработчик/TSS
0x00	Divide Error (DE, ошибка деления)	ntoskrnl.exe	KiTrap00
0x01	Debug (DB, отладка)	ntoskrnl.exe	KiTrap01
0x02	NMI Interrupt (прерывание NMI)	ntoskrnl.exe	KiNMITSS
0x03	BreakPoint (BP, точка останова)	ntoskrnl.exe	KiTrap03
0x04	Overflow (OF, переполнение)	ntoskrnl.exe	KiTrap04
0x05	BOUND Range Exceed (BR, выход за границы диапазона)	ntoskrnl.exe	KiTrap05
0x06	Undefined Opcode (UD, код операции не определен)	ntoskrnl.exe	KiTrap06
0x07	No Math Coprocessor (NM, нет сопроцессора)	ntoskrnl.exe	KiTrap07
0x08	Double Fault (DF, двойная ошибка)	ntoskrnl.exe	KiTrap08
0x09	Coprocessor Segment Overrun (выход за границы сегмента сопроцессора)	ntoskrnl.exe	KiTrap09
0x0A	Invalid TSS (TS, неверный TSS)	ntoskrnl.exe	KiTrap0A
0x0B	Segment Not Present (NP, сегмент отсутствует)	ntoskrnl.exe	KiTrap0B
0x0C	Stack Segment Fault (SS, ошибка сегмента стека)	ntoskrnl.exe	KiTrap0C
0x0D	General Protection (GP, общая ошибка защиты)	ntoskrnl.exe	KiTrap0D
0x0E	Page Fault (PF, ошибка страницы)	ntoskrnl.exe	KiTrap0E
0x0F	(зарезервировано Intel)	ntoskrnl.exe	KiTrap0F
0x10	Math Fault (MF, ошибка вычислений)	ntoskrnl.exe	KiTrap10

продолжение

Таблица 4.8. Шлюзы прерываний, ловушек и задач Windows 2000

INT	Описание INTEL	Владелец	Обработчик/TSS
0x11	Alignment Check (AC, проверка выравнивания)	ntoskrnl.exe	KiTrap11
0x12	Machine Check (MC, аппаратный сбой)	?	?
0x13	Streaming SIMD Extensions (потоковые расширения SIMD)	ntoskrnl.exe	KiTrap0F
0x14–0X1F	(зарезервировано Intel)	ntoskrnl.exe	KiTrap0F
0x2A	Определяется пользователем	ntoskrnl.exe	KiGetTickCount
0x2B	Определяется пользователем	ntoskrnl.exe	KiCallbackReturn
0x2C	Определяется пользователем	ntoskrnl.exe	KiSetLowWait-HighThread
0x2D	Определяется пользователем	ntoskrnl.exe	KiDebugService
0X2E	Определяется пользователем	ntoskrnl.exe	KiSystemService
0X2F	Определяется пользователем	ntoskrnl.exe	KiTrap0F
0x30	Определяется пользователем	hal.dll	HalpClockInterrupt
0x38	Определяется пользователем	hal.dll	HalpProfileInterrupt

Области памяти Windows 2000

Нам осталось обсудить последний параметр w2k_mem.exe, ключ +b. При выполнении команды с этим ключом генерируется чрезвычайно длинный список непрерывных областей памяти в пределах 4 Гбайт линейного адресного пространства. Программа создает этот список, проходя по всему расположенному по адресу 0xC0000000 массиву записей PTE при помощи функции IOCTL SPY_IO_PAGE_ENTRY драйвера слежения. Линейный адрес следующей записи PTE вычисляется на основе возвращенной этой функцией структуры SPY_PAGE_ENTRY, значение члена dSize которой прибавляется к линейному адресу, сопоставленному текущей записи PTE. В листинге 4.30 приведен соответствующий код.

Листинг 4.30. Поиск непрерывных блоков линейной памяти

```
DWORD WINAPI DisplayMemoryBlocks (HANDLE hDevice)
      {
      SPY_PAGE_ENTRY spe;
      PBYTE                pbPage, pbBase;
      DWORD                dBlock, dPresent, dTotal;
      DWORD                n = 0;

      pbPage         = 0;
      pbBase         = INVALID_ADDRESS;
      dBlock         = 0;
      dPresent       = 0;
      dTotal         = 0;

      n += _printf (L"\r\nContiguous memory blocks:"
                    L"\r\n-------------------------\r\n\r\n");

      do  {
          if (!IoControl (hDevice, SPY_IO_PAGE_ENTRY,
                          &pbPage, PVOID_,
```

```
                                &spe.     SPY_PAGE_ENTRY_))
            {
            n += _printf (L" !!! Device I/O error !!!\r\n");
            break;
            }
        if (spe.fPresent)
            {
            dPresent += spe.dSize;
            }
        if (spe.pe.dValue)
            {
            dTotal += spe.dSize;

            if (pbBase == INVALID_ADDRESS)
                {
                n += _printf (L"%5lu : 0x%08lX ->",
                              ++dBlock, pbPage);

                pbBase = pbPage;
                }
            }
        else
            {
            if (pbBase != INVALID_ADDRESS)
                {
              n += _printf (L" 0x%08lX (0x%08lX bytes)\r\n",
                            pbPage-1, pbPage-pbBase);

                pbBase = INVALID_ADDRESS;
                }
            }
        }
    while (pbPage += spe.dSize);

    if (pbBase != INVALID_ADDRESS)
        {
        n += _printf (L"0x%08lX\r\n", pbPage-1);
        }
    n += _printf (L"\r\n"
                  L" Present bytes: 0x%08lX\r\n"
                  L" Total   bytes: 0x%08lX\r\n",
                  dPresent, dTotal);
    return n;
    }
```

Пример 4.16 демонстрирует наиболее интересные области памяти, выбранные из результатов выполнения команды с этим параметром на моем компьютере. Сразу бросаются в глаза адреса 0x00400000, по которому расположен образ файла w2k_mem.exe (блок № 13), и 0x10000000, где находится образ w2k_lib.dll (блок № 23). Легко также найти страницы блоков TEB и PEB (блок № 104) и области памяти модулей hal.dll, ntoskrnl.exe и win32k.sys (блоки № 105 и 106). Блоки с № 340 по 350 — это, конечно, допустимые фрагменты массива PTE системы, в том числе и каталог страниц, занимающий часть блока № 347. В блоке № 2122 расположена область SharedUserData, а в блоке № 2123 — структуры KPCR, KPRCB и CONTEXT, содержащие информацию о состоянии потока и процессора.

Пример 4.16. Пример списка непрерывных блоков памяти

```
E:\>w2k_mem +b
[...]
Contiguous memory blocks:
-------------------------

     1 : 0x00010000 -> 0x00010FFF (0x00001000 bytes)
     2 : 0x00020000 -> 0x00020FFF (0x00001000 bytes)
     3 : 0x0012D000 -> 0x00138FFF (0x0000C000 bytes)
     4 : 0x00230000 -> 0x00230FFF (0x00001000 bytes)
     5 : 0x00240000 -> 0x00241FFF (0x00002000 bytes)
     6 : 0x00247000 -> 0x00247FFF (0x00001000 bytes)
     7 : 0x0024F000 -> 0x00250FFF (0x00002000 bytes)
     8 : 0x00260000 -> 0x00260FFF (0x00001000 bytes)
     9 : 0x00290000 -> 0x00290FFF (0x00001000 bytes)
    10 : 0x002E0000 -> 0x002E0FFF (0x00001000 bytes)
    11 : 0x002E2000 -> 0x002E3FFF (0x00002000 bytes)
    12 : 0x003B0000 -> 0x003B1FFF (0x00002000 bytes)
    13 : 0x00400000 -> 0x00404FFF (0x00005000 bytes)
    14 : 0x00406000 -> 0x00406FFF (0x00001000 bytes)
    15 : 0x00410000 -> 0x00410FFF (0x00001000 bytes)
    16 : 0x00419000 -> 0x00419FFF (0x00001000 bytes)
    17 : 0x0041B000 -> 0x0041BFFF (0x00001000 bytes)
    18 : 0x00450000 -> 0x00450FFF (0x00001000 bytes)
    19 : 0x00760000 -> 0x00760FFF (0x00001000 bytes)
    20 : 0x00770000 -> 0x00770FFF (0x00001000 bytes)
    21 : 0x00780000 -> 0x00783FFF (0x00004000 bytes)
    22 : 0x00790000 -> 0x00791FFF (0x00002000 bytes)
    23 : 0x10000000 -> 0x10003FFF (0x00004000 bytes)
    24 : 0x10005000 -> 0x10005FFF (0x00001000 bytes)
    25 : 0x1000E000 -> 0x10016FFF (0x00009000 bytes)
    26 : 0x759B0000 -> 0x759B1FFF (0x00002000 bytes)
[...]
   103 : 0x7FFD2000 -> 0x7FFD3FFF (0x00002000 bytes)
   104 : 0x7FFDE000 -> 0x7FFE0FFF (0x00003000 bytes)
   105 : 0x80000000 -> 0xA01A5FFF (0x201A6000 bytes)
   106 : 0xA01B0000 -> 0xA01F2FFF (0x00043000 bytes)
   107 : 0xA0200000 -> 0xA02C7FFF (0x000C8000 bytes)
   108 : 0xA02F0000 -> 0xA03FFFFF (0x00110000 bytes)
   109 : 0xA4000000 -> 0xA4001FFF (0x00002000 bytes)
   110 : 0xBE63B000 -> 0xBE63CFFF (0x00002000 bytes)
[...]
   340 : 0xC0000000 -> 0xC0001FFF (0x00002000 bytes)
   341 : 0xC0040000 -> 0xC0040FFF (0x00001000 bytes)
   342 : 0xC01D6000 -> 0xC01D6FFF (0x00001000 bytes)
   343 : 0xC01DA000 -> 0xC01DAFFF (0x00001000 bytes)
   344 : 0xC01DD000 -> 0xC01E0FFF (0x00004000 bytes)
   345 : 0xC01FD000 -> 0xC01FDFFF (0x00001000 bytes)
   346 : 0xC01FF000 -> 0xC0280FFF (0x00082000 bytes)
   347 : 0xC0290000 -> 0xC0301FFF (0x00072000 bytes)
   348 : 0xC0303000 -> 0xC0386FFF (0x00084000 bytes)
   349 : 0xC0389000 -> 0xC038CFFF (0x00004000 bytes)
   350 : 0xC039E000 -> 0xC03FFFFF (0x00062000 bytes)
[...]
  2121 : 0xFFC00000 -> 0xFFD0FFFF (0x00110000 bytes)
  2122 : 0xFFDF0000 -> 0xFFDF0FFF (0x00001000 bytes)
  2123 : 0xFFDFF000 -> 0xFFDFFFFF (0x00001000 bytes)
```

```
[...]
 Present bytes: 0x22AA9000
 Total   bytes: 0x2B8BA000
[...]
```

У параметра +b программы w2k_mem.exe есть одна своеобразная особенность: выводимый ей размер используемой памяти намного превышает любое разумное значение. Обратите внимание на строки итоговых значений в конце примера 4.16 — неужели я использую в данный момент 700 Мбайт памяти? Диспетчер задач показывает только 150 Мбайт, в чем же тут дело? Странный результат появился из-за блока № 105, который, судя по выданной информации, лежит в диапазоне адресов с 0x80000000 по 0xA01A5FFF, занимая 0x201A6000 байт, или, в десятичном формате, *538,599,434* байт. Это, конечно, полная чепуха. Проблема в том, что диапазон линейных адресов с 0x80000000 по 0x9FFFFFFF целиком отображается на диапазон физических адресов с 0x00000000 по 0x1FFFFFFF, как уже говорилось раньше в этой главе. Всем страницам объемом 4 Мбайт из этого диапазона соответствуют допустимые записи PDE в каталоге страниц по адресу 0xC0300000, что может подтвердить результат команды w2k_mem +d #0x20 0xC0300800 (пример 4.17). Поскольку все записи PDE в полученном списке — нечетные числа, соответствующие им страницы считаются присутствующими, но, несмотря на это, им не всегда соответствует физическая память. На самом деле большие фрагменты в этой области памяти представляют собой «дыры», которые при копировании в буфер будут содержать байты 0xFF. Так что не следует слишком серьезно воспринимать выводимое программой w2k_mem.exe итоговое значение используемой памяти.

Пример 4.17. Записи PDE для диапазона адресов 0x80000000–0x9FFFFFFF

```
E:\>w2k_mem +d #0x200 0xC0300800
[...]
C0300800..C03009FF: 512 valid bytes

Address  | 00000000 - 00000004 : 00000008 - 0000000C | 0000 0004 0008 000C
---------|---------------------:---------------------|-------------------
C0300800 | 000001E3 - 004001E3 : 008001E3 - 00C001E3 | ...r .@.r .?.r .A.r
C0300810 | 010001E3 - 014001E3 : 018001E3 - 01C001E3 | ...r .@.r .?.r .A.r
C0300820 | 020001E3 - 024001E3 : 028001E3 - 02C001E3 | ...r .@.r .?.r .A.r
C0300830 | 030001E3 - 034001E3 : 038001E3 - 03C001E3 | ...r .@.r .?.r .A.r
C0300840 | 040001E3 - 044001E3 : 048001E3 - 04C001E3 | ...r .@.r .?.r .A.r
C0300850 | 050001E3 - 054001E3 : 058001E3 - 05C001E3 | ...r .@.r .?.r .A.r
C0300860 | 060001E3 - 064001E3 : 068001E3 - 06C001E3 | ...r .@.r .?.r .A.r
C0300870 | 070001E3 - 074001E3 : 078001E3 - 07C001E3 | ...r .@.r .?.r .A.r
C0300880 | 080001E3 - 084001E3 : 088001E3 - 08C001E3 | ...r .@.r .?.r .A.r
C0300890 | 090001E3 - 094001E3 : 098001E3 - 09C001E3 | ...r .@.r .?.r .A.r
C03008A0 | 0A0001E3 - 0A4001E3 : 0A8001E3 - 0AC001E3 | ...r .@.r .?.r .A.r
C03008B0 | 0B0001E3 - 0B4001E3 : 0B8001E3 - 0BC001E3 | ...r .@.r .?.r .A.r
C03008C0 | 0C0001E3 - 0C4001E3 : 0C8001E3 - 0CC001E3 | ...r .@.r .?.r .A.r
C03008D0 | 0D0001E3 - 0D4001E3 : 0D8001E3 - 0DC001E3 | ...r .@.r .?.r .A.r
C03008E0 | 0E0001E3 - 0E4001E3 : 0E8001E3 - 0EC001E3 | ...r .@.r .?.r .A.r
C03008F0 | 0F0001E3 - 0F4001E3 : 0F8001E3 - 0FC001E3 | ...r .@.r .?.r .A.r
C0300900 | 100001E3 - 104001E3 : 108001E3 - 10C001E3 | ...r .@.r .?.r .A.r
C0300910 | 110001E3 - 114001E3 : 118001E3 - 11C001E3 | ...r .@.r .?.r .A.r
C0300920 | 120001E3 - 124001E3 : 128001E3 - 12C001E3 | ...r .@.r .?.r .A.r
```

продолжение

Пример 4.17 (*продолжение*)

```
C0300930 | 130001E3 - 134001E3 : 138001E3 - 13C001E3 | ...r .@.r .?.r .A.r
C0300940 | 140001E3 - 044001E3 : 148001E3 - 14C001E3 | ...r .@.r .?.r .A.r
C0300950 | 150001E3 - 154001E3 : 158001E3 - 15C001E3 | ...r .@.r .?.r .A.r
C0300960 | 160001E3 - 164001E3 : 168001E3 - 16C001E3 | ...r .@.r .?.r .A.r
C0300970 | 170001E3 - 174001E3 : 178001E3 - 17C001E3 | ...r .@.r .?.r .A.r
C0300980 | 180001E3 - 184001E3 : 188001E3 - 18C001E3 | ...r .@.r .?.r .A.r
C0300990 | 190001E3 - 194001E3 : 198001E3 - 19C001E3 | ...r .@.r .?.r .A.r
C03009A0 | 1A0001E3 - 1A4001E3 : 1A8001E3 - 1AC001E3 | ...r .@.r .?.r .A.r
C03009B0 | 1B0001E3 - 1B4001E3 : 1B8001E3 - 1BC001E3 | ...r .@.r .?.r .A.r
C03009C0 | 1C0001E3 - 1C4001E3 : 1C4001E3 - 1CC001E3 | ...r .@.r .?.r .A.r
C03009D0 | 1D0001E3 - 1D4001E3 : 1D8001E3 - 1DC001E3 | ...r .@.r .?.r .A.r
C03009E0 | 1E0001E3 - 1E4001E3 : 1E8001E3 - 1EC001E3 | ...r .@.a .?.a .A.a
C03009F0 | 1F0001E3 - 1F4001E3 : 1F8001E3 - 1FC001E3 | ...a .@.a .?.a .A.a
[...]
```

Карта памяти Windows 2000

В последней части главы будет представлена общая схема разделения 4 Гбайт линейного адресного процесса так, как ее «видит» процесс Windows 2000. В табл. 4.9 приведены диапазоны адресов многих весьма важных структур данных. Длинные промежутки памяти между ними используются в различных целях, например как области загрузки модулей процесса и драйверов устройств, пулов памяти, списков рабочих множеств и т. п. Отметьте, что в разных системах некоторые адреса и размеры блоков могут значительно различаться в зависимости от конфигурации памяти и оборудования, свойств процесса и некоторых других переменных. Так что пользуйтесь этим списком только как приблизительной схемой, а не как точной детальной картой.

Некоторые блоки физической памяти появляются в линейном адресном пространстве два или более раз. Например, область SharedUserData по линейному адресу 0xFFDF0000 зеркально отображается по адресу 0x7FFE0000. Обе области соответствуют одной и той же странице в физической памяти — при записи байта данных по адресу 0xFFDF0000+n загадочным образом изменяется значение байта по адресу 0x7FFE0000+n. Это обычное дело в мире виртуальной памяти — физический адрес может отображаться в любом месте линейного адресного пространства, даже по нескольким адресам одновременно. Все зависит от соответствующей установки каталога страниц и таблиц страниц. Вернитесь, пожалуйста, к рис. 4.3 и 4.4, которые ясно демонстрируют фиктивность линейных адресов. Их битовые поля Directory и Table — всего лишь указатели на структуры, в которых указано настоящее место расположения данных. Если же номера PFN двух записей PTE оказались одинаковыми, это значит, что соответствующие линейные адреса относятся к одному и тому же месту в физической памяти.

Таблица 4.9. Распознаваемые области памяти в адресном пространстве процесса

Начало	Конец	Размер (HEX)	Тип/Описание
0x00000000	0x0000FFFF	10000	Нижний предохранительный блок
0x00010000	0x0001FFFF	10000	WCHRA[]/строки окружения, размещенные в страницах объемом 4 Кбайт

Начало	Конец	Размер (HEX)	Тип/Описание
0x00020000	0x0002FFFF	10000	PROCESS_PARAMETERS/размещенные в страницах объемом 4 Кбайт
0x00030000	0x0012FFFF	100000	DWORD[4000]/стек процесса (по умолчанию 1 Мбайт)
0x7FFDD000	0x7FFDDFFF	1000	TEB/блок переменных окружения потока для потока № 1
0x7FFDE000	0x7FFDEFFF	1000	TEB/блок переменных окружения потока для потока № 2
0x7FFDF000	0x7FFDFFFF	1000	PEB/блок переменных окружения процесса
0x7FFE0000	0x7FFE02D7	2D8	KUSER_SHARED_DATA/SharedUserData в пользовательском режиме
0x7FFF0000	0x7FFFFFFF	10000	Верхний предохранительный блок
0x80000000	0x800003FF	400	IVT/Interrupt Vector Table, таблица векторов прерываний
0x80036000	0x800363FF	400	KGDTENTRY[80]/глобальная таблица дескрипторов
0x80036400	0x80036BFF	800	KIDTENTRY[100]/таблица дескрипторов прерываний
0x800C0000	0x800FFFFF	40000	VGA/ROM BIOS
0x80244000	0x802460AA	20AB	KTSS/сегмент состояния задачи (занят) режима пользователя/ядра
0x8046AB80	0x8046ABBF	40	KeServiceDescriptorTable
0x8046ABC0	0x8046ABBF	40	KeServiceDescriptorTableShadow
0x80470040	0x804700A7	68	KTSS/KiDoubleFaultTSS
0x804700A8	0x8047010F	68	KTSS/KiNMITSS
0x804704D8	0x804708B7	3E0	PROC[F8]/KiServiceTable
0x804708B8	0x804708BB	4	DWORD/KiServiceLimit
0x804708BC	0x804709B3	F8	BYTE[F8]/KiArgumentTable
0x814C6000	0x82CC5FFF	1800000	PFN[100000]/MmPfnDatabase (максимально для 4 Гбайт)
0xA01859F0	0xA01863EB	9FC	PROC[27F]/W32pServiceTable
0xA0186670	0xA01868EE	27F	BYTE[27F]/W32pArgumentTable
0xC0000000	0xC03FFFFF	400000	X86_PE[100000]/каталог страниц и таблицы страниц
0xC1000000	0xE0FFFFFF	20000000	Системный кэш (MmSystemCacheStart, MmSystemCacheEnd)
0xE1000000	0xE77FFFFF	6800000	Страничный пул (MmPagedPoolStart, MmPagedPoolEnd)
0xF0430000	0xF043FFFF	10000	Сегмент кода ROM BIOS
0xF0440000	0xF044FFFF	10000	Сегмент данных ROM BIOS
0xFFDF0000	0xFFDF02D7	2D8	KUSER_SHARED_DATA/SharedUserData в режиме ядра
0xFFDFF000	0xFFDFF053	54	KPCR/область управляющего блока процессора (сегмент FS в режиме ядра)
0xFFDFF120	0xFFDFF13B	1C	KPRCB/управляющий блок процессора
0xFFDFF13C	0xFFDFF407	2CC	CONTEXT/контекст потока (состояние процессора)
0xFFDFF620	0xFFDFF71F	100	Каталоги списков просмотра

5 Слежение за вызовами функций Native API

Перехват вызовов операционной системы всегда был любимым занятием всех программистов. Для этого есть ряд причин: профилирование и оптимизация кода, обратное проектирование, запись действий пользователя и т. п. Чтобы обеспечить все эти потребности, необходимо решить одну общую задачу: при вызове системной функции из приложения нужно передать управление особому участку кода, который сможет определить вызванную системную функцию, переданные ей параметры, возвращаемый результат и время выполнения. В этой главе рассмотрена общая схема внедрения перехватчиков в произвольные функции Native API, основанная на технике, впервые предложенной Марком Руссиновичем (Mark Russinovich) и Брюсом Когсвеллом (Bryce Cogswell), см. Russinovich and Cogswell, 1997. Используемый здесь подход полностью управляется данными, поэтому его можно легко расширить и модифицировать для других версий Windows 2000/NT. Собранные данные о вызовах API из всех процессов в системе записываются в кольцевой буфер, доступный клиентскому приложению через устройство управления вводом-выводом. Данные представляют собой разбитый на строки поток символов ANSI, подчиняющийся строгим правилам форматирования, поэтому процесс его последующей обработки в вызывающем приложении не составляет труда. В качестве базовой схемы в этой главе будет рассмотрено консольное приложение для просмотра данных отчета.

Модификация таблицы дескрипторов системных вызовов

В «примитивных» операционных системах, таких как DOS или Windows 3.xx, программистам не составляло большого труда справиться с защитой системы, для того чтобы установить перехватчики функций прикладных программных интер-

фейсов (Application Programming Interface, API). Сделать это в системах Win32, таких как Windows 2000, Windows NT и Windows 9x, значительно сложнее, поскольку в них применяются продуманные защитные механизмы отделения друг от друга независимых программ. Установка общесистемного перехватчика для Win32 API — весьма трудоемкая задача. К счастью, есть такие крупные специалисты по Win32, как Матт Пиетрек (Matt Pietrek) и Джеффри Рихтер (Jeffrey Richter), которые подробно описали ее реализацию, несмотря на отсутствие простого и элегантного решения. В 1997 г. Руссинович и Когсвелл представили совершенно другой подход к установке общесистемных перехватчиков для Windows NT, работая с системой на гораздо более низком уровне. Они предложили внедрить механизм записи в диспетчер интерфейса Native API до границы между уровнями пользователя и ядра, — в том месте, где Windows NT создает «пропускной пункт», через который обязаны пройти все потоки пользовательского режима при обращении к ядру операционной системы.

Таблица аргументов и таблица системных вызовов

Как обсуждалось в главе 2, все вызовы Native API из пользовательского режима должны пройти по одному и тому же маршруту — через интерфейс INT 2Eh, реализующий шлюз прерываний процессора i386 для смены уровня привилегий. Как вы помните, все вызовы INT 2Eh обрабатываются в режиме ядра внутренней функцией KiSystemService(), которая ищет точки входа обработчиков функций Native API в таблице дескрипторов системных вызовов (Service Descriptor Table, SDT). Рисунок 5.1 иллюстрирует внутренние взаимосвязи основных частей этого механизма. В листинге 5.1 повторно приведены формальные определения структуры SERVICE_DESCRIPTOR_TABLE и производные от нее типы из главы 2 (листинг 2.1).

KiSystemService() принимает два аргумента, которые вызывающая ее по INT 2Eh функция передает через регистры процессора EAX и EDX. Регистр EAX содержит индекс (индексация с нуля) в массиве указателей на функции-обработчики API, а EDX указывает на стек аргументов вызывающей функции. KiSystemService() получает базовый адрес массива функций, считывая значение члена ServiceTable открытой структуры данных KeServiceDescriptorTable из модуля ntoskrnl.exe (рис. 5.1). В действительности KeServiceDescriptorTable указывает на массив из четырех структур, определяющих параметры таблицы вызовов, но по умолчанию только первая из них содержит допустимые записи. Функция KiSystemService() ищет адрес функции-обработчика вызова API, используя значение регистра EAX в качестве индекса внутренней структуры KiServiceTable. Перед вызовом требуемой функции KiSystemService() аналогичным образом обращается к структуре KiArgumentTable, получая количество байт передаваемых аргументов в стеке аргументов вызывающей функции и копируя их в текущий стек режима ядра. После этого обработчик API вызывается простой ассемблерной инструкцией CALL. Затем все устанавливается в значения, ожидаемые обычной функцией C, использующей соглашение о вызовах __stdcall.

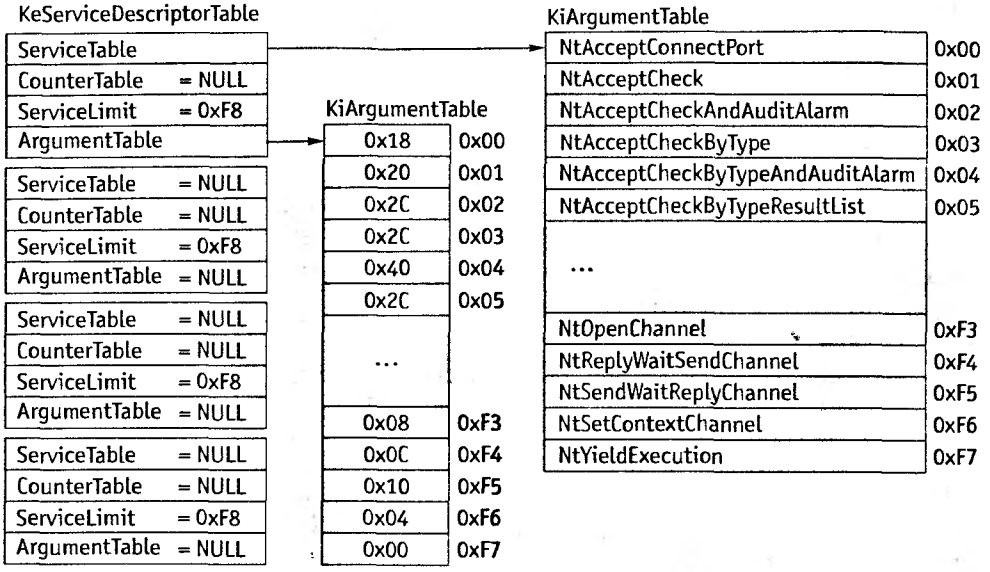

Рис. 5.1. Схема KeServiceDescriptorTable

Листинг 5.1. Определение структуры SERVICE_DESCRIPTOR_TABLE

```
typdef NTSTATUS (NTAPI *NTPROC) ();
typdef NTPROC *PNTPROC;
#define NTPROC_ sizeof (NTPROC)

//  ---------------------------------------------------------
typedef struct _SYSTEM_SERVICE_TABLE
      {
      PNTPROC   ServiceTable;          // массив точек входа
      PDWORD    CounterTable;          // массив счетчиков
                                       // использования
      DWORD     ServiceLimit;          // число элементов
                                       // в таблице
      PBYTE     ArgumentTable;         // массив счетчиков
                                           // байт
      }
          SYSTEM_SERVICE_TABLE,
      * PSYSTEM_SERVICE_TABLE,
      **PPSYSTEM_SERVICE_TABLE;

//  ---------------------------------------------------------

typedef struct _SERVICE_DESCRIPTOR_TABLE
      {
 SYSTEM_SERVICE_TABLE  ntoskrnl;       // ntoskrnl.exe (native API)
      SYSTEM_SERVICE_TABLE  win32k;    // win32k.sys (поддержка gdi/user)
      SYSTEM_SERVICE_TABLE  Table3;    // не используется
      SYSTEM_SERVICE_TABLE  Table4;    // не используется
      }
          SERVICE_DESCRIPTOR_TABLE,
      * PSERVICE_DESCRIPTOR_TABLE,
      **PPSERVICE_DESCRIPTOR_TABLE;
```

В Windows 2000 существует еще один блок параметров таблицы дескрипторов по имени KeServiceDescriptorTableShadow. В отличие от KeServiceDescriptorTable, которая открыто экспортируется ntoskrnl.exe и непосредственно доступна дайверам режима ядра, KeServiceDescriptorTableShadow таковой не является. В Windows 2000 KeServiceDescriptorTableShadow расположен сразу за KeServiceDescriptorTable, но не стоит всегда на это расчитывать — в Windows NT 4.0 это не так, и, возможно, в следующих за Windows 2000 системах это тоже будет неверно. Различие между двумя блоками параметров состоит в том, что в KeServiceDescriptorTableShadow вторая запись также используется системой. Она содержит ссылки на внутренние структуры w32pServiceTable и w32pArgumentTable, которые используются компонентом Win32 режима ядра w32.sys для обработки своих собственных вызовов API, как показано на рис. 5.2 Функция KiSystemService() определяет, что ей нужно обработать вызов API win32k.sys, по значениям битов № 12 и № 13 индекса функции в регистре EAX. Равенство обоих битов нулю свидетельствует о вызове Native API из модуля ntoskrnl.exe, поэтому KiSystemService() использует первую запись в таблице SDT. Если бит № 12 установлен, а бит № 13 — равен нулю, KiSystemService() использует вторую запись. Две другие пары значений бит соответствуют двум оставшимся записям, что в данный момент системой не используется. Это означает, что значения индексов вызовов Native API потенциально могут принимать значения из диапазона 0x0000–0x0FF, а индексы вызовов win32k.sys — из диапазона 0x1000–0x1FFF. Соответственно диапазоны 0x2000–0x2FFF и 0x3000–0x3FFF выделены для зарезервированных таблиц. В Windows 2000 таблица вызовов Native API состоит из 248 записей, а таблица win32k.sys — из 639 записей.

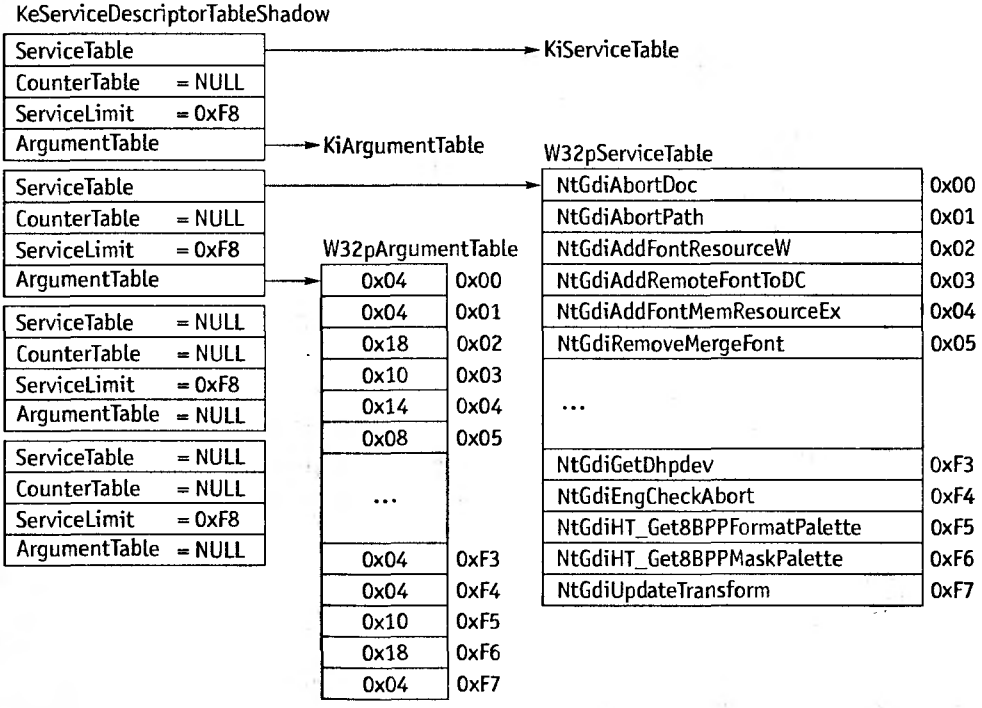

Рис. 5.2. Схема KeServiceDescriptorTableShadow

Оригинальная идея Руссиновича и Когсвелла заключалась в том, чтобы перехватывать вызовы API путем простой замены обработчиков в массиве KiServiceTable. Установленный взамен обработчик, разумеется, вызывал бы первоначальный из модуля ntoskrnl.exe, но у нового обработчика появлялась возможность просматривать входные и выходные параметры вызываемой функции. Такой подход исключительно эффективен и к тому же очень прост. Поскольку все потоки пользовательского режима вынужденно проходят через это игольное ушко, чтобы получить результат своих запросов к Native API, простая замена указателей на функции устанавливает глобальный перехватчик, надежно работающий даже после запуска новых процессов и потоков. Нет необходимости в каком-либо дополнительном механизме, который бы уведомлял о добавлении и удалении процессов и потоков.

К сожалению, таблицы указателей на системные вызовы зачастую весьма нетривиально изменяются от версии к версии Windows NT. В табл. 5.1 сравниваются записи в таблицах KiServiceTable Windows 2000 и Windows NT 4.0. Помимо того что число обработчиков выросло с 211 до 248, можно заметить, что новые обработчики были не просто добавлены в конец, а вставлены в различные места по всему списку! Таким образом, например, значение 0x20 индекса системной функции в случае Windows 2000 ссылается на NtCreateFile(), а в Windows NT 4.0 ему соответствует NtCreateProfile(). Следовательно, монитор вызовов API, устанавливающий перехватчик путем изменения записей в таблице системных функций, должен обязательно проверить версию Windows NT, в которой он выполняется. Это можно сделать несколькими способами:

- один способ — проверить значение открытой переменной NtBuildNumber, экспортируемой ntoskrnl.exe, как это сделали Руссиновичи Когсвелл в исходной статье. Windows NT 4.0 выдает номер сборки 1381 для всех пакетов обновлений. Номер сборки Windows 2000 в данный момент равен 2195. Будем надеяться, что этот номер продержится так же долго, как и в предыдущих версиях Windows;
- другой способ — проверять значения членов NtMajorVersion и NtMinorVersion структуры SharedUserData, определенной в заголовочном файле Windows 2000 ntddk.h. Для всех пакетов обновлений Windows NT значение SharedUserData->NtMajorVersion равно четырем, а значение SharedUserData->NtMinorVersion равно нулю. Windows 2000 сейчас обозначает версию Windows NT как 5.0;
- программа, представленная в этой главе, использует еще одну альтернативу — она проверяет значение члена ServiceLimit записи SDT, которое равно 211 (0xD3) для Windows NT 4.0 и 248 (0xF8) для Windows 2000.

Таблица 5.1. Сравнение таблиц системных вызовов NT 4.0 и Windows 2000

Windows 2000	Индекс	Windows NT 4.0
NtAcceptConnectPort	0x00	NtAcceptConnectPort
NtAccessCheck	0x01	NtAccessCheck
NtAccessCheckAndAuditAlarm	0x02	NtAccessCheckAndAuditAlarm
NtAccessCheckByType	0x03	NtAddAtom
NtAccessCheckByTypeAndAuditAlarm	0x04	NtAdjustGroupsToken

Windows 2000	Индекс	Windows NT 4.0
NtAccessCheckByTypeResultList	0x05	NtAdjustPrivilegesToken
NtAccessCheckByTypeResultListAndAuditAlarm	0x06	NtAlertResumeThread
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x07	NtAlertThread
NtAddAtom	0x08	NtAllocateLocallyUniquId
NtAdjustGroupsToken	0x09	NtAllocateUuids
NtAdjustPrivilegesToken	0x0A	NtAllocateVirtualMemory
NtAlertResumeThread	0x0B	NtCallbackReturn
NtAlertThread	0x0C	NtCancelIoFile
NtAllocateLocallyUniqueId	0x0D	NtCancelTimer
NtAllocateUserPhysicalPages	0x0E	NtClearEvent
NtAllocateUuids	0x0F	NtClose
NtAllocateVirtualMemory	0x10	NtCloseObjectAuditAlarm
NtAreMappedFilesTheSame	0x11	NtCompleteConnectPort
NtAssignProcessToJobObject	0x12	NtConnectPort
NtCallbackReturn	0x13	NtContinue
NtCancelIoFile	0x14	NtCreateDirectoryObject
NtCancelTimer	0x15	NtCreateEvent
NtCancelDeviceWakeupRequest	0x16	NtCreateEventPair
NtClearEvent	0x17	NtCreateFile
NtClose	0x18	NtCreateIoCompletion
NtCloseObjectAuditAlarm	0x19	NtCreateKey
NtCompleteConnectPort	0x1A	NtCreateMailslotFile
NtConnectPort	0x1B	NtCreateMutant
NtContinue	0x1C	NtCreateNamedPipeFile
NtCreateDirectoryObject	0x1D	NtCreatePagingFile
NtCreateEvent	0x1E	NtCreatePort
NtCreateEventPair	0x1F	NtCreateProcess
NtCreateFile	0x20	NtCreateProfile
NtCreateIoCompletion	0x21	NtCreateSection
NtCreateJobObject	0x22	NtCreateSemaphore
NtCreateKey	0x23	NtCreateSymbolicLinkObject
NtCreateMailslotFile	0x24	NtCreateThread
NtCreateMutant	0x25	NtCreateTimer
NtCreateNamedPipeFile	0x26	NtCreateToken
NtCreatePagingFile	0x27	NtDelayExecution
NtCreatePort	0x28	NtDeleteAtom
NtCreateProcess	0x29	NtDeleteFile
NtCreateProfile	0x2A	NtDeleteKey
NtCreateSection	0x2B	NtDeleteObjectAuditAlarm
NtCreateSemaphore	0x2C	NtDeleteValueKey
NtCreateSymbolicLinkObject	0x2D	NtDeviceIoControlFile
NtCreateThread	0x2E	NtDisplayString

продолжение

Таблица 5.1 (*продолжение*)

Windows 2000	Индекс	Windows NT 4.0
NtCreateTimer	0x2F	NtDuplicateObject
NtCreateToken	0x30	NtDuplicateToken
NtCreateWaitablePort	0x31	NtEnumerateKey
NtDelayExecution	0x32	NtEnumerateValueKey
NtDeleteAtom	0x33	NtExtendSection
NtDeleteFile	0x34	NtFindAtom
NtDeleteKey	0x35	NtFlushBuffersFile
NtDeleteObjectAuditAlarm	0x36	NtFlushInstructionCache
NtDeleteValueKey	0x37	NtFlushKey
NtDeviceIoControlFile	0x38	NtFlushVirtualMemory
NtDisplayString	0x39	NtFlushWriteBuffer
NtDuplicateObject	0x3A	NtFreVirtualMemory
NtDuplicateToken	0x3B	NtFsControlFile
NtEnumerateKey	0x3C	NtGetContextThread
NtEnumerateValueKey	0x3D	NtGetPlugPlayEvent
NtExtendSection	0x3E	NtGetTickCount
NtFilterToken	0x3F	NtImpersonateClientOfPort
NtFindAtom	0x40	NtImpersonateThread
NtFlushBuffersFile	0x41	NtInitializeRegistry
NtFlushInstructionCache	0x42	NtListenPort
NtFlushKey	0x43	NtLoadDriver
NtFlushVirtualMemory	0x44	NtLoadKey
NtFlushWriteBuffer	0x45	NtLoadKey2
NtFreeUserPhysicalPages	0x46	NtLockFile
NtFreVirtualMemory	0x47	NtLockVirtualMemory
NtFsControlFile	0x48	NtMakeTemporaryObject
NtGetContextThread	0x49	NtMapViewOfSection
NtGetDevicePowerState	0x4A	NtNotifyChangeDirectoryFile
NtGetPlugPlayEvent	0x4B	NtNotifyChangeKey
NtGetTickCount	0x4C	NtOpenDirectoryObject
NtGetWriteWatch	0x4D	NtOpenEvent
NtImpersonateAnonymousToken	0x4E	NtOpenEventPair
NtImpersonateClientOfPort	0x4F	NtOpenFile
NtImpersonateThread	0x50	NtOpenIoCompletion
NtInitializeRegistry	0x51	NtOpenKey
NtInitializePowerAction	0x52	NtOpenMutant
NtIsSystemResumeAutomatic	0x53	NtOpenObjectAuditAlarm
NtListenPort	0x54	NtOpenProcess
NtLoadDriver	0x55	NtOpenProcessToken
NtLoadKey	0x56	NtOpenSection
NtLoadKey2	0x57	NtOpenSemaphore

Windows 2000	Индекс	Windows NT 4.0
NtLockFile	0x58	NtOpenSymbolicLinkObject
NtLockVirtualMemory	0x59	NtOpenThread
NtMakeTemporaryObject	0x5A	NtOpenThreadToken
NtMapUserPhysicalPages	0x5B	NtOpenTimer
NtMapUserPhysicalPagesScatter	0x5C	NtPlugPlayControl
NtMapViewOfSection	0x5D	NtPrivilegeCheck
NtNotifyChangeDirectoryFile	0x5E	NtPrivilegedServiceAuditAlarm
NtNotifyChangeKey	0x5F	NtPrivilegeObjectAuditAlarm
NtNotifyChangeMultipleKey	0x60	NtProtectVirtualMemory
NtOpenDirectoryObject	0x61	NtPulseEvent
NtOpenEvent	0x62	NtQueryInformationAtom
NtOpenEventPair	0x63	NtQueryAttributesFile
NtOpenFile	0x64	NtQueryDefaultLocale
NtOpenIoCompletion	0x65	NtQueryDirectoryFile
NtOpenJobObject	0x66	NtQueryDirectoryObject
NtOpenKey	0x67	NtQueryEaFile
NtOpenMutant	0x68	NtQueryEvent
NtOpenObjectAuditAlarm	0x69	NtQueryFullAtributesFile
NtOpenProcess	0x6A	NtQueryInformationFile
NtOpenProcessToken	0x6B	NtQueryIoCompletion
NtOpenSection	0x6C	NtQueryInformationPort
NtOpenSemaphore	0x6D	NtQueryInformationProcess
NtOpenSymbolicLinkObject	0x6E	NtQueryInformationThread
NtOpenThread	0x6F	NtQueryInformationToken
NtOpenThreadToken	0x70	NtQueryIntervalProfile
NtOpenTimer	0x71	NtQueryKey
NtPlugPlayControl	0x72	NtQueryMultipleValueKey
NtPowerInformation	0x73	NtQueryMutant
NtPrivilegeCheck	0x74	NtQueryObject
NtPrivilegedServiceAudItAlarm	0x75	NtQueryOleDirectoryFile
NtPrivilegeObjectAuditAlarm	0x76	NtQueryPerformanceCounter
NtProtectVirtualMemory	0x77	NtQuerySection
NtPulseEvent	0x78	NtQuerySecurityObject
NtQueryInformationAtom	0x79	NtQuerySemaphore
NtQueryAttributesFile	0x7A	NtQuerySymbolicLinkObject
NtQueryDefaultLocale	0x7B	NtQuerySystemEnvironmentValue
NtQueryDefaultUILanguage	0x7C	NtQuerySystemInformation
NtQueryDirectoryFile	0x7D	NtQuerySystemTime
NtQueryDirectoryObject	0x7E	NtQueryTimer
NtQueryEaFile	0x7F	NtQueryTimerResolution
NtQueryEvent	0x80	NtQueryValueKey
NtQueryFullAtributesFile	0x81	NtQueryVirtualMemory

продолжение

Таблица 5.1 (*продолжение*)

Windows 2000	Индекс	Windows NT 4.0
NtQueryInformationFile	0x82	NtQueryVolumeInformationFile
NtQueryInformationJobObject	0x83	NtQueueApcThread
NtQueryIoCompletion	0x84	NtRaiseException
NtQueryInformationPort	0x85	NtRaiseHardError
NtQueryInformationProcess	0x86	NtReadFile
NtQueryInformationThread	0x87	NtReadFileScatter
NtQueryInformationToken	0x88	NtReadRequestData
NtQueryInstallUILanguage	0x89	NtReadVirtualMemory
NtQueryIntervalProfile	0x8A	NtRegisterThreadTerminatePort
NtQueryKey	0x8B	NtReleaseMutant
NtQueryMultipleValueKey	0x8C	NtReleaseSemaphore
NtQueryMutant	0x8D	NtRemoveIoCompletion
NtQueryObject	0x8E	NtReplaceKey
NtQueryOpenSubKeys	0x8F	NtReplyPort
NtQueryPerformanceCounter	0x90	NtReplyWaitReceivePort
NtQueryQuotaInformationFile	0x91	NtReplyWaitReplyPort
NtQuerySection	0x92	NtRequestPort
NtQuerySecurityObject	0x93	NtRequestWaitReplyPort
NtQuerySemaphore	0x94	NtResetEvent
NtQuerySymbolicLinkObject	0x95	NtRestoreKey
NtQuerySystemEnvironmentValue	0x96	NtResumeThread
NtQuerySystemInformation	0x97	NtSaveKey
NtQuerySystemTime	0x98	NtSetIoCompletion
NtQueryTimer	0x99	NtSetContextThread
NtQueryTimerResolution	0x9A	NtSetDefaultHardErrorPort
NtQueryValueKey	0x9B	NtSetDefaultLocale
NtQueryVirtualMemory	0x9C	NtSetEaFile
NtQueryVolumeInformationFile	0x9D	NtSetEvent
NtQueueApcThread	0x9E	NtSetHighEventPair
NtRaiseException	0x9F	NtSetHighWaitLowEventPair
NtRaiseHardError	0xA0	NtSetHighWaitLowThread
NtReadFile	0xA1	NtSetInformationFile
NtReadFileScatter	0xA2	NtSetInformationKey
NtReadRequestData	0xA3	NtSetInformationObject
NtReadVirtualMemory	0xA4	NtSetInformationProcess
NtRegisterThreadTerminatePort	0xA5	NtSetInformationThread
NtReleaseMutant	0xA6	NtSetInformationToken
NtReleaseSemaphore	0xA7	NtSetIntervalProfile
NtRemoveIoCompletion	0xA8	NtSetLdtEntries
NtReplaceKey	0xA9	NtSetLowEventPair
NtReplyPort	0xAA	NtSetLowWaitHighEventPair

Windows 2000	Индекс	Windows NT 4.0
NtReplyWaitReceivePort	0xAB	NtSetLowWaitHighThread
NtReplyWaitReceivePortEx	0xAC	NtSetSecurityObject
NtReplyWaitReplyPort	0xAD	NtSetSystemEnvironmentValue
NtRequestDeviceWakeup	0xAE	NtSetSystemInformation
NtRequestPort	0xAF	NtSetSystemPowerState
NtRequestWaitReplyPort	0xB0	NtSetSystemTime
NtRequestWakeupLatency	0xB1	NtSetTimer
NtResetEvent	0xB2	NtSetTimerResolution
NtResetWriteWatch	0xB3	NtSetValueKey
NtRestoreKey	0xB4	NtSetVolumeInformationFile
NtResumeThread	0xB5	NtShutdownSystem
NtSaveKey	0xB6	NtSignalAndWaitForSingleObject
NtSaveMergedKeys	0xB7	NtStartProfile
NtSecureConnectPort	0xB8	NtStopProfile
NtSetIoCompletion	0xB9	NtSuspendThread
NtSetContextThread	0xBA	NtSystemDebugControl
NtSetDefaultHardErrorPort	0xBB	NtTerminateProcess
NtSetDefaultLocale	0xBC	NtTerminateThread
NtSetDefaultUILanguage	0xBD	NtTestAlert
NtSetEaFile	0xBE	NtUnloadDriver
NtSetEvent	0xBF	NtUnloadKey
NtSetHighEventPair	0xC0	NtUnlockFile
NtSetHighWaitLowEventPair	0xC1	NtUnlockVirtualMemory
NtSetInformationFile	0xC2	NtUnmapViewOfSection
NtSetInformationJobObject	0xC3	NtVdmControl
NtSetInformationKey	0xC4	NtWaitForMultipleObjects
NtSetInformationObject	0xC5	NtWaitForSingleObject
NtSetInformationProcess	0xC6	NtWaitHighEventPair
NtSetInformationThread	0xC7	NtWaitLowEventPair
NtSetInformationToken	0xC8	NtWriteFile
NtSetIntervalProfile	0xC9	NtWriteFileGather
NtSetLdtEntries	0xCA	NtWriteRequestData
NtSetLowEventPair	0xCB	NtWriteVirtualMemory
NtSetLowWaitHighEventPair	0xCC	NtCreateChannel
NtSetQuotaInformationFile	0xCD	NtListenChannel
NtSetSecurityObject	0xCE	NtOpenChannel
NtSetSystemEnvironmentValue	0xCF	NtReplyWaitSendChannel
NtSetSystemInformation	0xD0	NtSendWaitReplyChannel
NtSetSystemPowerState	0xD1	NtSetContextChannel
NtSetSystemTime	0xD2	NtYieldExecution
NtSetThreadExecutionState	0xD3	отсутствует
NtSetTimer	0xD4	отсутствует

продолжение

Таблица 5.1 (*продолжение*)

Windows 2000	Индекс	Windows NT 4.0
NtSetTimerResolution	0xD5	отсутствует
NtSetUuidSeed	0xD6	отсутствует
NtSetValueKey	0xD7	отсутствует
NtSetVolumeInformationFile	0xD8	отсутствует
NtShutdownSystem	0xD9	отсутствует
NtSignalAndWaitForSingleObject	0xDA	отсутствует
NtStartProfile	0xDB	отсутствует
NtStopProfile	0xDC	отсутствует
NtSuspendThread	0xDD	отсутствует
NtSystemDebugControl	0xDE	отсутствует
NtTerminateJobObject	0xDF	отсутствует
NtTerminateProcess	0xE0	отсутствует
NtTerminateThread	0xE1	отсутствует
NtTestAlert	0xE2	отсутствует
NtUnloadDriver	0xE3	отсутствует
NtUnloadKey	0xE4	отсутствует
NtUnlockFile	0xE5	отсутствует
NtUnlockVirtualMemory	0xE6	отсутствует
NtUnmapViewOfSection	0xE7	отсутствует
NtVdmControl	0xE8	отсутствует
NtWaitForMultipleObjects	0xE9	отсутствует
NtWaitForSingleObject	0xEA	отсутствует
NtWaitHighEventPair	0xEB	отсутствует
NtWaitLowEventPair	0xEC	отсутствует
NtWriteFile	0xED	отсутствует
NtWriteFileGather	0xEE	отсутствует
NtWriteRequestData	0xEF	отсутствует
NtWriteVirtualMemory	0xF0	отсутствует
NtCreateChannel	0xF1	отсутствует
NtListenChannel	0xF2	отсутствует
NtOpenChannel	0xF3	отсутствует
NtReplyWaitSendChannel	0xF4	отсутствует
NtSendWaitReplyChannel	0xF5	отсутствует
NtSetContextChannel	0xF6	отсутствует
NtYieldExecution	0xF7	отсутствует

Наиболее важным шагом Руссиновича и Когсвелла было решение о создании драйвера устройства режима ядра, который бы устанавливал и поддерживал работу перехватчиков Native API, поскольку модули режима пользователя не обладают соответствующими привилегиями для изменения системы на таком низком уровне. Аналогично драйверу просмотра памяти из главы 4, это весьма необычный драйвер, в котором не реализована типичная обработка запросов ввода-вывода.

Через механизм управления вводом-выводом устройства (Device I/O Control, IOCTL) в драйвере реализован только элементарный интерфейс, обеспечивающий доступ к собранным драйвером данным для кода пользовательского режима. Главным образом драйвер выполняет операции с KiServiceTable и осуществляет перехват и протоколирование определенных вызовов Windows 2000 Native API. Хотя этот способ прост и элегантен, он также внушает некоторые опасения. Его простота напоминает мне старые времена работы с DOS, когда для перехвата системного вызова было достаточно просто изменить указатель в таблице векторов прерываний процессора. Любой, кто знает, как написать драйвер режима ядра для Windows 2000, сможет без особых усилий перехватить любой системный вызов NT.

При помощи этой техники Руссинович и Когсвелл создали чрезвычайно удобный монитор реестра NT. При модифицикации их кода для других задач наблюдения мне быстро надоело создавать отдельную функцию перехвата для каждой наблюдаемой функции API, как требовалась по их методике. Чтобы не писать практически одинаковый код, я задался целью поместить все интересующие меня функции API в одну функцию перехвата. При решении этой оказавшейся весьма трудоемкой задачи я имел удовольствие познакомиться со всеми вариантами синих экранов смерти (BSOD). Тем не менее в результате было выработано общее решение, которое позволило без больших сложностей менять набор перехватываемых функций API.

Реабилитация ассемблера

Основное препятствие при создании общего решения возникало из-за стандартного механизма передачи параметров языка C. Как вы знаете, перед передачей управления точке входа функции компилятор языка C обычно передает аргументы функции в стек процессора. В зависимости от числа аргументов функции, размер этого стека меняется в широких пределах. Для 248 функций Native API Windows 2000 размер стека аргументов меняется от нуля до 68 байт. Из-за постоянной строгой проверки типов в C создание единой функции перехвата потребовало бы очень больших усилий. В поставку Microsoft Visual C/C++ входит многоцелевой встроенный компилятор языка ассемблера (ASM), способный обрабатывать код умеренной сложности. По иронии, преимущество ассемблера в данной ситуации заключается как раз в том, что считается одним из его самых больших недостатков: в отсутствии механизма строгой проверки типов. При согласовании числа бит можно хранить практически что угодно в любых регистрах и вызывать подпрограмму по любому адресу, не заботясь о том, что в данный момент находится в стеке. Хотя такая свобода опасна при создании прикладных программ, в этой ситуации она очень удобна: в ассемблере можно беспрепятственно переходить на общую точку входа с различными аргументами в стеке, что я и реализовал в рассматриваемом диспетчере перехватов вызовов API.

Чтобы вызвать встроенный ассемблер в Microsoft Visual C/C++, нужно поместить ассемблерный код в блоки, предваряемые ключевым словом __asm. Во встроенном ассемблере не поддерживаются средства работы с макросами языка Macro Assembler (MASM), но это не очень сильно ограничивает его функциональность. Наиболее привлекательная особенность встроенного ассемблера — это то, что ему доступны

все переменные и определения типов языка C, поэтому смешивать код на C с кодом встроенного ассемблера очень легко. Однако при вставке кода ASM в функцию C необходимо соблюдать некоторые важные соглашения компилятора языка C, чтобы не повлиять на скомпилированный код C:

- вызвавшая функцию C программа считает, что значения регистров процессора EBP, ESI и EDI не изменяются;
- если код ассемблера используется вместе с кодом на C в одной функции, необходимо сохранять все промежуточные значения кода C, которые могут находиться в регистрах. Всегда неплохо сохранять и восстанавливать значения всех регистров, используемых в пределах вставки __asm;
- 8-битные возвращаемые значения функций (CHAR, BYTE и т. д.) возвращаются в регистре AL;
- 16-битные возвращаемые значения функций (SHORT, WORD и т. д.) возвращаются в регистре AX;
- 32-битные возвращаемые значения функций (LONG, DWORDLONG, указатели и т. д.) возвращаются в регистре EAX;
- 64-битные возвращаемые значения функций (__int64, LONGLONG, DWORDLONG и т. д.) возвращаются в паре регистров EDX:EAX. В регистре EAX содержатся биты с № 0 по № 31, а в регистре EDX — биты с № 32 по № 63;
- функции с фиксированным числом аргументов обычно передают аргументы в соответствии с соглашением о вызовах __stdcall. С точки зрения вызывающей программы это означает, что перед вызовом аргументы должны помещаться в стек в обратном порядке, а очистку стека должна осуществлять вызываемая функция. С точки зрения вызываемой функции это означает, что указатель стека ESP указывает на адрес возврата вызывающей программы, вслед за которым расположены агрументы в своем естественном порядке. Этот порядок сохраняется, поскольку стек растет вниз, в сторону уменьшения линейных адресов. Поэтому аргумент, помещенный вызывающей функцией в стек последним (то есть аргумент № 1), будет первым аргументом в массиве, на который указывает регистр ESP;
- некоторые функции API с фиксированным количеством аргументов, главным образом функциии библиотеки времени выполнения C (C Runtime Library), экспортируемые ntdll.dll и ntoskrnl, традиционно придерживаются соглашения о передаче параметров __cdecl. В этом случае порядок аргументов такой же, как и у __stdcall, но стек должна очищать вызывающая функция;
- функции с переменным числом аргументов всегда придерживаются соглашения __cdecl, поскольку только вызывающая функция знает точное число переданных аргументов. Поэтому ответственность за удаление аргументов из стека возложена на вызывающую функцию;
- функции, объявленные с модификатором __fastcall, ожидают первых два аргумента в регистрах процессора ECX и EDX. Последующие аргументы передаются через стек в обратном порядке, и стек очищает вызывающая функция, как в соглашении __stdcall;

- многие компиляторы языка C сразу же после входа в функцию создают стековый кадр для аргументов функции, используя регистр-указатель базы процессора EBP. Генерируемый для этого код, показанный в листинге 5.2, часто называют «прологом» и «эпилогом» функции. Некоторые компиляторы применяют более тонкие операции i386 ENTER и LEAVE, объединяющие обмен регистров EBP/ESP в одной инструкции. После выполнения пролога стек выглядит так, как показано на рис. 5.3. Значение регистра EBP становится единой точкой отсчета, разделяющей стек параметров функции на область локальных данных (1), в которой хранятся все локальные переменные, определенные в области видимости функции, и стек аргументов вызывающей функции (2), включающий сохраненное значение EBP и адрес возврата. Заметьте, что последние версии Microsoft Visual C/C++ по умолчанию не используют стековые кадры. Доступ к значениям в стеке осуществляется только через регистр ESP при помощи указания смещения переменной относительно текущей вершины стека. Читать такой код — просто кошмар, поскольку каждая инструкция PUSH и POP меняет значение ESP, а следовательно, и все смещения параметров. Поскольку в этой схеме регистр EBP не участвует, он используется как дополнительный регистр общего назначения;

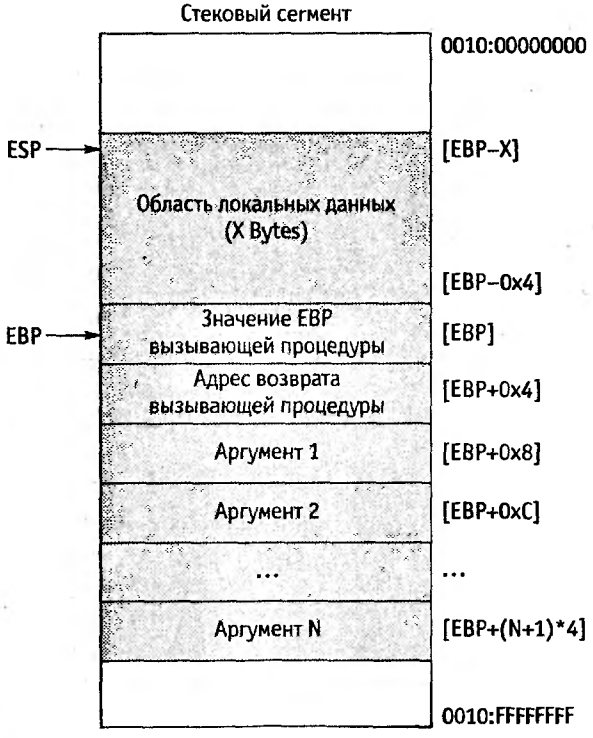

Рис. 5.3. Типичный стековый кадр

Листинг 5.2. Стековый кадр, пролог и эпилог

```
; пролог функции
push ebp                              ; сохранить текущее значение ebp
mov      ebp. esp                     ; установить адрес базы стекового кадра
sub      esp. SizeOfLocalStorage      ; создать область локальных данных
...
; эпилог функции
mov      esp. ebp                     ; освободить область локальных данных
push ebp                              ; восстановить значение ebp
ret
```

- уделяйте повышенное внимание работе с переменными С. При работе со встроенным ассемблером очень часто встречается ошибка, когда в регистр загружается адрес переменной вместо ее значения, и наоборот. В случае возможной неоднозначности используйте операции с адресами ptr и offset. Например, инструкция mov eax, dword ptr SomeVariable загружает в регистр EAX значение переменной SomeVariable типа DWORD в отличие от инструкции mov eax, offset SomeVariable, загружающей в регистр EAX линейный адрес переменной (то есть указатель на ее значение).

Диспетчер перехватов

Следующий код чрезвычайно сложен. Я писал его много часов, постоянно сталкиваясь с BSOD. Сначала я создал отдельный модуль на чистом языке ассемблера и скомпилировал его при помощи Microsoft MASM. Однако при таком подходе возникали проблемы на этапе компоновки, поэтому я перешел к применению встроенного ассемблера, вызываемого из основного модуля С. Я решил не создавать еще один драйвер режима ядра, а интегрировать код перехвата вызовов в драйвер слежения, описанный в главе 4. Помните группу функций IOCTL SPY_IO_HOOK_*, приведенных в нижней части табл. 4.2? Рассмотрим их теперь подробнее. Следующая часть программ-примеров взята из файлов исходного кода w2k_spy.c и w2k_spy.h, расположенных в каталоге \src\w2k_spy прилагающегося к книге компакт-диска.

Ключевые части механизма перехвата функций Native API показаны в листинге 5.3. Вначале приведен ряд определений констант и структур, после которых — определение массива aSpyHooks[]. Вслед за ним определена макроподстановка, разворачивающаяся в три важные строки встроенного ассемблера, которые будут рассмотрены чуть позже. В последней части листинга 5.3 приведено определение функции SpyHookInitializeEx(). На первый взгляд не так-то просто понять, что она делает. Эта функция состоит из двух других:

1. «Внешняя» часть SpyHookInitializeEx() состоит из кода на С, который просто заполняет массив aSpyHooks[] указателями на функции перехвата и связанные с ними строки, определяющие формат протокола. Эта функция разделена на две части. Первая часть заканчивается внутри первого предложения __asm на инструкции jmp SpyHook9. Вторая часть, очевидно, должна начинаться с ассемблерной метки SpyHook9, которая расположена в конце второго блока __asm.
2. «Внутренняя» часть SpyHookInitializeEx() содержит все строки между двумя секциями кода на С. Она начинается с многочисленных вызовов макроса SpyHook, за которыми следует длинный и сложный фрагмент кода на ассемблере. Как можно догадаться, этот код и есть тот общий обработчик перехватов, о котором говорилось раньше.

Листинг 5.3. Реализация диспетчера перехватов

```
#define SPY_CALLS               0x00000100      // максимальный уровень
                                                // вложенности вызовов API
#define SDT_SYMBOLS_NT4   0xD3
#define SDT_SYMBOLS_NT5   0xF8
#define SDT_SYMBOLS_MAX   SDT_SYMBOLS_NT5

//  ---------------------------------------------------------

typedef struct _SPY_HOOK_ENTRY
      {
      NTPROC    Handler;
      PBYTE     pbFormat;
      }
      SPY_HOOK_ENTRY, *PSPY_HOOK_ENTRY, **PPSPY_HOOK_ENTRY;

#define SPY_HOOK_ENTRY_ sizeof (SPY_HOOK_ENTRY)

//  ---------------------------------------------------------

typedef struct _SPY_CALL
      {
      BOOL              fInUse;               // установлен, если запись использовалась
      HANDLE            hThread;              // идентификатор вызывающего потока
      PSPY_HOOK_ENTRY   pshe;                 // соответствующая запись перехватчика
      PVOID             pCaller;              // адрес возврата вызывающей функции
      DWORD             dParameters;          // количество параметров
      DWORD             adParameters [1+256]; // результат и параметры
      }
      SPY_CALL, *PSPY_CALL, **PPSPY_CALL;

#define SPY_CALL_ sizeof (SPY_CALL)

//  ---------------------------------------------------------

SPY_HOOK_ENTRY aSpyHooks [SDT_SYMBOLS_MAX];

//  ---------------------------------------------------------

// Макрос SpyHook определяет точку входа в обработчик
// на встроенном ассемблере.
// Вход в общую точку входа SpyHook2 осуществляется
// инструкцией call, что позволяет определить перехватчик
// по его адресу возврата в стеке.
// Вызов осуществляется через регистр, для того чтобы
// убрать из кодирования вызова любую неоднозначность.

#define SpyHook                                         \
          __asm     push eax                          \
          __asm     mov      eax, offset SpyHook2     \
          __asm     call eax

//  ---------------------------------------------------------
// Функция SpyHookInitializeEx() инициализирует
// массив aSpyHooks[] точками входа перехватчиков и
// форматирующими строками. В ней также содержатся точки
// входа перехватчиков и диспетчер перехвата.
```

продолжение ⇩

Листинг 5.3 (*продолжение*)

```
void SpyHookInitializeEx (PPBYTE  ppbSymbols,
                          PPBYTE  ppbFormats)
      {
      DWORD dHooks1, dHooks2, i, j, n;

      __asm
          {
          jmp       SpyHook9
          ALIGN     8
SpyHook1:       ; начало раздела точек входа перехватчиков
          }

// число определенных в этом разделе точек входа
// должно быть равно SDT_SYMBOLS_MAX (то есть 0xF8)

SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //08
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //10
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //18
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //20
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //28
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //30
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //38
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //40
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //48
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //50
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //58
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //60
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //68
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //70
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //78
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //80
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //88
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //90
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //98
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F8
      __asm
          {
SpyHook2:  ; конец раздела точек входа перехватчиков
          pop       eax                          ; удалить адрес возврата
          pushfd
          push      ebx
          push      ecx
          push      edx
          push      ebp
          push      esi
```

```
        push    edi
        sub     eax, offset SpyHook1        ; вычислить индекс
                                            ; точки входа
        mov     ecx, SDT_SYMBOLS_MAX
        mul     ecx
        mov     ecx, offset SpyHook2
        sub     ecx, offset SpyHook1
        div     ecx
        dec     ax
        mov     ecx, gfSpyHookPause         ; флаг проверки паузы
        add     ecx, -1
        sbb     ecx, ecx
        not     ecx
        lea     edx, [aSpyHooks + eax * SIZE SPY_HOOK_ENTRY]
        test    ecx, [edx.pbFormat]         ; форматирующая строка
                                            ; == NULL?
        jz      SpyHook5
        push    eax
        push    edx
        call    PsGetCurrentThreadId        ; получить
                                            ; идентификатор потока
        mov     ebx, eax
        pop     edx
        pop     eax
        cmp     ebx, ghSpyHookThread        ; не учитывать
                                            ; установщик перехватчика
        jz      SpyHook5
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]         ; получить массив
                                            ; контекстов вызовов
        mov     esi, SPY_CALLS              ; получить количество
                                            ; записей
SpyHook3:
        mov     ecx, 1                      ; установить флаг
                                            ; использования
        xchg    ecx, [edi.fInUse]
        jecxz   SpyHook4                    ; найдена неиспользуемая
                                            ; запись
        add     edi, SIZE SPY_CALL          ; перейти к следующей
                                            ; записи
        dec     esi
        jnz     SpyHook3
        mov     edi, gpDeviceContext
        inc     [edi.dMisses]               ; подсчитать несовпадения
        jmp     SpyHook5                    ; переполнение массива
SpyHook4:
        mov     esi, gpDeviceContext
        inc     [esi.dLevel]                ; установить уровень
                                            ; вложенности
        mov     [edi.hThread], ebx          ; сохранить идентификатор
                                            ; потока
        mov     [edi.pshe], edx             ; сохранить
                                            ; PSPY_HOOK_ENTRY
        mov     ecx, offset SpyHook6        ; установить новый
                                            ; адрес возврата
        xchg    ecx, [esp+20h]
```

продолжение

Листинг 5.3 (*продолжение*)

```
        mov     [edi.pCaller], ecx          ; сохранить старый адрес
                                            ; возврата
        mov     ecx, KeServiceDescriptorTable
        mov     ecx, [ecx].ntoskrnl.ArgumentTable
        movzx   ecx, byte ptr [ecx+eax]     ; получить размер
                                            ; стека аргументов
        shr     ecx, 2
        inc     ecx                         ; добавить 1 для
                                            ; записи результата
        mov     [edi.dParameters], ecx      ; сохранить число
                                            ; параметров
        lea     edi, [edi.adParameters]
        xor     eax, eax                    ; инициализировать
                                            ; позицию результата
        stosd
        dec     ecx
        jz      SpyHook5                    ; нет аргументов
        lea     esi, [esp+24h]              ; сохранить стек
                                            ; аргументов
        rep     movsd
SpyHook5:
        mov     eax, [edx.Handler]          ; получить первоначальный обработчик
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]                  ; восстановить eax и...
        ret                                 ; ...перейти на
                                            ; обработчик
SpyHook6:
        push    eax
        pushfd
        push    ebx
        push    ecx
        push    edx
        push    ebp
        push    esi
        push    edi
        push    eax
        call    PsGetCurrentThreadId        ; получить идентификатор потока
        mov     ebx, eax
        pop     eax
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]         ; получить массив
                                            ; контекстов вызовов
        mov     esi, SPY_CALLS              ; получить количество записей
SpyHook7:
        cmp     ebx, [edi.hThread]          ; найти нужный
                                            ; идентификатор потока
        jz      SpyHook8
        add     edi, SIZE SPY_CALL          ; перейти к следующей записи
        dec     esi
```

```
        jnz     SpyHook7
        push    ebx                             ; запись не найдена ?!?
        call    KeBugCheck
SpyHook8:
        push    edi                             ; сохранить указатель
                                                ; на SPY_CALL
        mov     [edi.adParameters], eax         ; записать NTSTATUS
        push    edi
        call    SpyHookProtocol
        pop     edi                             ; восстановить указатель
                                                ; на SPY_CALL
        mov     eax, [edi.pCaller]
        mov     [edi.hThread], 0                ; очистить идентификатор
                                                ; потока
        mov     esi, gpDeviceContext
        dec     [esi.dLevel]                    ; восстановить уровень
                                                ; вложенности
        dec     [edi.fInUse]                    ; очистить флаг
                                                ; использования
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]                      ; восстановить eax и...
        ret                                     ; ...вернуться в
                                                ; вызывающую функцию
SpyHook9:
        mov     dHooks1, offset SpyHook1
        mov     dHooks2, offset SpyHook2
        }
    n = (dHooks2 - dHooks1) / SDT_SYMBOLS_MAX;

    for (i = j = 0; i < SDT_SYMBOLS_MAX;
                        i++, dHooks1 += n)
        {
        if ((ppbSymbols   != NULL) && (ppbFormats != NULL)
        &&
            (ppbSymbols [j] != NULL))
            {
            aSpyHooks [i].Handler  = (NTPROC) dHooks1;
            aSpyHooks [i].pbFormat =
                SpySearchFormat (ppbSymbols [j++],
                                 ppbFormats);
            }
        else
            {
            aSpyHooks [i].Handler  = NULL;
            aSpyHooks [i].pbFormat = NULL;
            }
        }
    return;
    }
```

Что же представляет собой макрос SpyHook? В теле функции SpyHookInitializeEx() он повторяется в точности 248 (0xF8) раз, что совпадает с количеством функций Native API Windows 2000. В верхней части листинга 5.3 в это значение устанавливается константа SDT_SYMBOLS_MAX, которая равна максимуму из SDT_SYMBOLS_NT4 и SDT_SYMBOLS_NT5. Да, именно так — я собираюсь поддерживать также и Windows NT 4.0! Вернемся к макросу SpyHook: эта последовательность вызовов реализована в ассемблерном коде, показанном в листинге 5.4. Каждый макрос SpyHook разворачивается в три строки кода:

1. Сначала текущее значение регистра EAX сохраняется в стеке.
2. Затем в регистр EAX записывается линейный адрес метки SpyHook2.
3. Наконец, выполняется инструкция CALL — вызов по адресу из регистра EAX.

А что будет при возврате из этого вызова? Будет ли исполняться следующая группа строк SpyHook? Нет, этого не будет — возврат из этого вызова CALL не предусмотрен, поскольку после перехода на метку SpyHook2 адрес возврата сразу же удаляется из стека инструкцией POP EAX (листинг 5.4). Этот на первый взгляд бессмысленный код был придуман в старые добрые времена программирования на ассемблере, в наши дни повсеместной объектно-ориентированной разработки приложений на языках высокого уровня этот прием вышел из употребления. Такая техника применялась ассемблерными гуру, когда требовалось создать массив однородных точек входа, каждой из которых соответствовала бы своя функция. Практически одинаковый код для всех точек входа гарантирует равные интервалы, поэтому индекс используемой клиентом точки входа легко можно вычислить, зная адрес возврата инструкции CALL, базовый адрес и общий размер массива, а также количество элементов в массиве, по простому правилу трех.

Листинг 5.4. Подстановка при встрече SpyHook

```
SpyHook1:
        push    eax
        mov             eax, Offset SpyHook2
        call    eax
        push    eax
        mov             eax, Offset SpyHook2
        call    eax
: пропущено 244 однообразных фрагментов кода
        push    eax
        mov             eax, Offset SpyHook2
        push    eax
        mov             eax, Offset SpyHook2
        call    eax
SpyHook2:
        pop             eax
```

Например, адрес возврата первой инструкции CALL EAX в листинге 5.4 — это адрес второй точки входа. В общем случае адрес возврата N-го вызова CALL EAX равен адресу элемента N+1, за исключением последнего вызова, который, естественно, должен осуществить возврат к SpyHook2. Таким образом, индекс массива (индексация с нуля) всех точек входа можно вычислить по общей формуле, показанной на рис. 5.4. Правило трех заключается в следующем: точки входа в количестве

SDT_SYMBOLS_MAX расположены в блоке памяти SpyHook2 - SpyHook1. Сколько точек входа расположено в блоке ReturnAddress – SpyHook1? Поскольку результат вычисления по этой формуле лежит между единицей и SDT_SYMBOLS_MAX, его нужно уменьшить на единицу для получения правильного индекса массива.

Реализацию формулы из рис. 5.4 можно найти в листинге 5.3, сразу же после ассемблерной метки SpyHook2. Этот код также показан в левом нижнем углу рис. 5.5, на котором приведен общий принцип работы механизма распределения перехватов. Заметьте, что инструкция MUL процессора i386 возвращает 64-битный результат в регистрах EDX:EAX, а инструкция DIV ожидает 64-битное делимое также в регистрах EDX:EAX, поэтому нет риска целочисленного переполнения. В левом верхнем углу изображена таблица KiServiceTable, для которой макрос SpyHook сгенерирует адреса точек входа. В середине рисунка снова показан развернутый код макроса из листинга 5.4. Линейные адреса точек входа показаны справа. По случайному совпадению размер каждой точки входа равен 8 байтам, поэтому адрес вычисляется путем умножения на 8 индекса функции в таблице KiServiceTable и прибавлению этой величины к адресу SpyHook1.

$$\text{Index} = \frac{(\text{ReturnAddress}-\text{SpyHook1})*\text{SDT\_SYMBDLS\_MAX}}{\text{SpyHook2}-\text{SpyHook1}} - 1$$

Рис. 5.4. Определение точки входа перехватчика по его адресу возврата

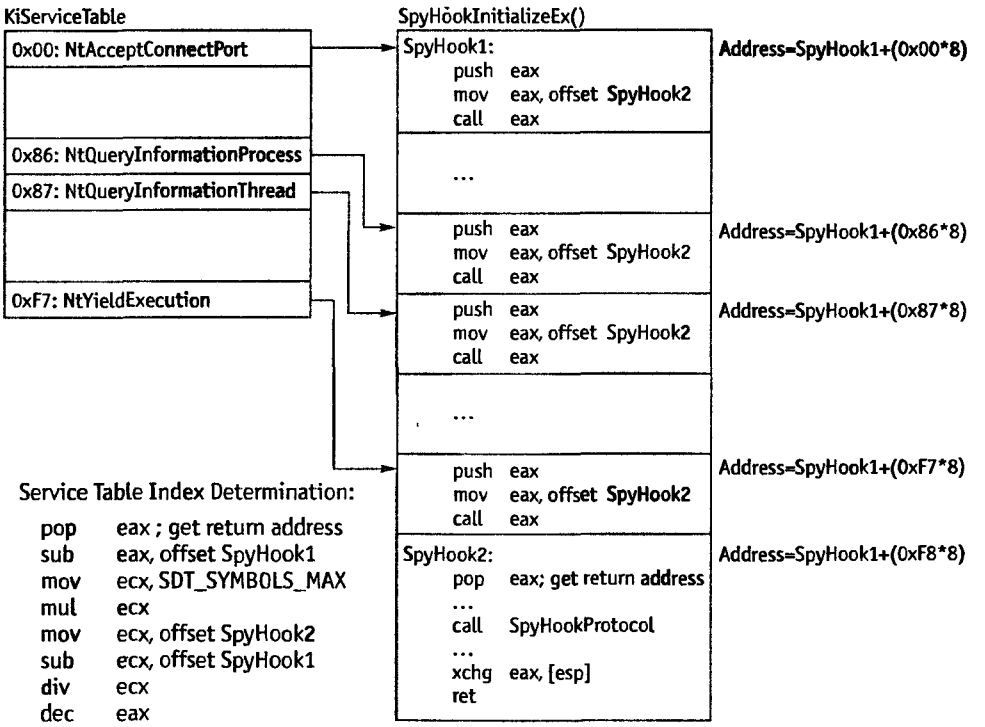

Рис. 5.5. Принцип действия диспетчера перехватов

На самом деле я просто пошутил — размер каждой точки входа, равный 8 байтам, это *не* случайное совпадение. Я посвятил немало времени разработке наилучшей реализации точек входа перехватов. Хотя в этом нет жесткой необходимости, выравнивание кода по границам 32 бит никогда не помешает, поскольку это повышает производительность. Конечно, в данном случае выигрыш в производительности небольшой. Вы можете спросить, зачем я делаю косвенный вызов CALL на метку SpyHook2 через регистр EAX — не будет ли значительно эффективней применить стандартную инструкцию CALL SpyHook2? Будет! Но проблема инструкций процессора i386 call и jump в том, что они могут быть реализованы по-разному, действуя одинаково, но обладая различным размером инструкции. Материал по этой теме можно найти в документации Intel «Instruction Set Reference» для семейства процессоров Pentium.

Поскольку выбор используемого варианта зависит от компилятора/ассемблера, нет гарантии, что для всех точек входа будет сгенерирован одинаковый код. Инструкция же MOV EAX с постоянным 32-битным операндом всегда кодируется одинаково, так же как и инструкция CALL EAX.

Следует пояснить еще некоторые места листинга 5.3. Давайте начнем с завершающего раздела кода на C, начинающегося после метки SpyHook9. В ассемблерном коде после этой метки переменным dHook1 и dHook2 предварительно присваиваются значения линейных адресов меток SpyHook1 и SpyHook2. Затем в переменную n записывается размер точки входа в перехватчик, который вычисляется путем деления размера массива точек входа на число его элементов. Конечно, это значение будет равно восьми. Оставшаяся часть листинга 5.3 представляет собой цикл, в котором инициализируются все элементы глобального массива aSpyHooks[]. Массив состоит из структур SPY_HOOK_ENTRY, определение которых приведено в верхней половине листинга 5.3, каждому элементу массива соответствует функция Native API. Чтобы понять, как присваиваются значения членам структуры Handler и pbFormat, нужна дополнительная информация о передаваемых функции SpyHookInitializeEx() параметрах ppbSymbols и ppbFormats. В листинге 5.5 показана функция-оболочка SpyHookInitialize(), вызывающая SpyHookInitializeEx() с аргументами, соответствующими текущей версии операционной системы. Как говорилось раньше, код не проверяет версию операционной системы или номер сборки напрямую, а сравнивает значение члена ServiceLimit соответствующей ntoskrnl.exe записи в таблице SDT с константами SDT_SYMBOLS_NT4 и SDT_SYMBOLS_NT5. Если значение не равно ни одной из этих констант, драйвер слежения инициализирует все элементы массива aSpyHook[] NULL-указателями, полностью отключая, таким образом, механизм перехвата функций Native API.

Листинг 5.5. Выбор идентификатора, соответствующего текущей версии операционной системы в функции SpyHookInitialize()

```
BOOL SpyHookInitialize (void)
     {
     BOOL fOk = TRUE.

     switch  (KeServiceDescriptorTable->ntoskrnl ServiceLimit)
         {
```

```
        case SDT_SYMBOLS_NT4:
             {
             SpyHookInitializeEx  (apbSdtSymbolsNT4,
                                   apbSdtFormats).
             break.
             }
        case SDT_SYMBOLS_NT5.
             {
             SpyHookInitializeEx  (apbSdtSymbolsNT5.
                                   apbSdtFormats).
             break.
             }
        default.
             {
             SpyHookInitializeEx (NULL, NULL).
             fOk = FALSE;
             break.
             }
        }
    return fOk;
    }
```

Передаваемые в качестве первого параметра ppbSymbols функции SpyHookInitializeEx() глобальные массивы apbSdtSymbolsNT4[] и apbSdtSymbolsNT5[] представляют собой простые таблицы строк, содержащие все имена функций Native API Windows NT 4.0 и Windows 2000, отсортированные по своему индексу KiServiceTable и заканчивающиеся NULL-указателем. Массив строк apbSdtFormats[] показан в листинге 5.6. Этот список форматных строк — одна из наиболее важных частей механизма перехвата, поскольку именно от него зависит, какие вызовы Native API будут отслеживаться и как будет выглядеть каждая запись в отчете. Сразу видно, что схема построения этих строк похожа на схему функции printf() из библиотеки времени выполнения C, только формат строк специально приспособлен к наиболее часто используемым типам аргументов функций Native API. В табл. 5.2 приведен полный список форматирующих спецификаторов, которые распознает регистратор вызовов API.

Листинг 5.6. Форматные строки регистратора вызовов Native API

```
PBYTE apbSdtFormats [] =
    {
    "%s=NtCancelIoFile(%!.%1)",
    "%s=NtClose(%-)",
    "%s=NtCreateFile(%+.%n %o.%1.%l.%n.%n.%n.%n.%p.%n)",
    "%s=NtCreateKey(%+.%n.%o.%n.%u.%n.%d)",
    "%s=NtDeleteFile(%o)",
    "%s=NtDeleteKey(%-)",
    "%s=NtDeleteValueKey(%! %u)",
    "%s=NtDeviceIoControlFile(%!.%p.%p.%p.%1.%n %p.%n.%p.%n)",
    "%s=NtEnumerateKey(%! %n.%n.%p.%n.%d)",
    "%s=NtEnumerateValueKey(%!.%n.%n.%p.%n.%d)",
    "%s=NtFlushBuffersFile(%!.%1)",
    "%s=NtFlushKey(%!)",
    "%s=NtFsControlFile(%!.%p.%p.%p.%1.%n.%p.%n.%p.%n)",
```

продолжение

Листинг 5.6 (*продолжение*)

```
"%s=NtLoadKey(%o.%o)",
"%s=NtLoadKey2(%o.%o.%n)",
"%s=NtNotifyChangeKey(%!.%p.%p.%p.%i.%n.%b.%p.%n.%b)",
"%s=NtNotifyChangeMultipleKeys(%!.%n.%o.%p.%p.%p.%i.%n.%b.%p.%n.%b)",
"%s=NtOpenFile(%+.%n.%o.%i.%n.%n)",
"%s=NtOpenKey(%+.%n.%o)",
"%s=NtOpenProcess(%+.%n.%o.%c)",
"%s=NtOpenThread(%+.%n.%o.%c)",
"%s=NtQueryDirectoryFile(%!.%p.%p.%p.%i.%p.%n.%n.%b.%u.%b)",
"%s=NtQueryInformationFile(%!.%i.%p.%n.%n)",
"%s=NtQueryInformationProcess(%!.%n.%p.%n.%d)",
"%s=NtQueryInformationThread(%!.%n.%p.%n.%d)",
"%s=NtQueryKey(%!.%n.%p.%n.%d)",
"%s=NtQueryMultipleValueKey(%!.%p.%n.%p.%d.%d)",
"%s=NtQueryOpenSubKeys(%o.%d)",
"%s=NtQuerySystemInformation(%n.%p.%n.%d)",
"%s=NtQuerySystemTime(%l)",
"%s=NtQueryValueKey(%!.%u.%n %p.%n.%d)",
"%s=NtQueryVolumeInformationFile(%!.%i.%p.%n.%n)",
"%s=NtReadFile(%!.%p.%p.%p.%i.%p.%n.%l.%d)",
"%s=NtReplaceKey(%o.%!.%o)",
"%s=NtSetInformationKey(%!.%n.%p.%n)",
"%s=NtSetInformationFile(%!.%i.%p.%n.%n)",
"%s=NtSetInformationProcess(%!.%n.%p.%n)",
"%s=NtSetInformationThread(%!.%n.%p.%n)",
"%s=NtSetSystemInformation(%n.%p.%n)",
"%s=NtSetSystemTime(%l.%l)",
"%s=NtSetValueKey(%!.%u.%n.%n.%p.%n)",
"%s=NtSetVolumeInformationFile(%!.%i.%p.%n.%n)",
"%s=NtUnloadKey(%o)",
"%s=NtWriteFile(%!.%p.%p.%p.%i.%p.%n.%l.%d)",
NULL
};
```

Важно отметить, что имя функции в каждой форматной строке должно быть записано правильно. SpyHookInitializeEx() проходит по списку идентификаторов функций Native API, который передается ей через параметр ppbSymbols, и пытается найти в списке ppbFormats форматную строку, содержащую искомое имя функции. Сравнение происходит при помощи вспомогательной функции SpySearchFormat(), вызываемой в операторе if в конце листинга 5.3. Поскольку для задания значений всем элементам массива aSpyHook[] необходимо произвести большое число операций поиска строк, для этого я использую высокооптимизированный поисковый механизм, основанный на оригинальном «алгоритме поиска Shift/And». Дополнительную информацию о его реализации можно почерпнуть из исходного кода группы функций SpySearch*() в файле \src\w2k_spy\w2k_spy.c на прилагающемся к книге компакт-диске. После выхода из цикла в функции SpyHookInitializeEx() все члены Handler в массиве aSpyHook[] указывают на надлежащие точки входа перехватчиков, а члены pbFormat, если это требуется, сдержат соответствующие форматные строки. В случае Windows NT 4 оба члена у элементов массива в диапазоне индексов с 0xD3 по 0xF8 установлены в NULL, поскольку для этой версии они не определены.

Таблица 5.2. Управляющие спецификаторы формата

Вид	Название	Описание
%+	Дескриптор (зарегистрировать)	Записывает дескриптор и имя объекта, добавляет их в таблицу дескрипторов
%!	Дескриптор (извлечь)	Записывает дескриптор и извлекает имя соответствующего ему объекта из таблицы дескрипторов
%-	Дескриптор (отменить регистрацию)	Записывает дескриптор и имя объекта и удаляет их из таблицы дескрипторов
%a	строка ANSI	Записывает строку 8-битных символов ANSI
%b	BOOLEAN	Записывает 8-битное значение типа BOOLEAN
%c	CLIENT_ID *	Записывает значения членов структуры CLIENT_ID
%d	DWORD *	Записывает значение DWORD по указанному адресу
%I	IO_STATUS_BLOCK *	Записывает значения членов структуры IO_STATUS_BLOCK
%I	LARGE_INTEGER	Записывает значение структуры LARGE_INTEGER
%n	Число (DWORD)	Записывает значение 32-битного числа без знака
%o	OBJECT_ATRIBUTES *	Записывает ObjectName объекта
%p	Указатель	Записывает соответствующий указателю адес
%s	Состояние (NTSTATUS)	Записывает код состояния NT
%u	UNICODE_STRING *	Записывает значение члена Buffer структуры UNICODE_STRING
%w	Строка широких символов	Записывает строку 16-битных символов Unicode
%%	ESC-последовательность для символа процента	Записывает один символ '%'

Наиболее примечательное свойство схемы работы механизма перехвата — его полная управляемость данными. Диспетчер перехвата можно видоизменить для нового выпуска Windows 2000, просто добавив новую таблицу идентификаторов API. Более того, протоколирование работы дополнительных функций API можно включить в любое время, добавив в массив apbSdtFormats[] новые форматные строки. Не требуется писать какой-либо дополнительный код, работа драйвера слежения за функциями API полностью определяется набором строк символов! При определении форматных строк, тем не менее, следует быть осторожным. Никогда не упускайте из виду, что w2k_spy.sys работает как драйвер режима ядра. На системном уровне ошибки обрабатываются не очень тщательно. При работе с Win32 API задание неверных аргументов функции не страшно — будет выведено окно с сообщением об ошибке, и приложение завершится. В режиме ядра малейшая ошибка нарушения доступа приведет к появлению BSOD. Будьте осторожны — несоответствующий или пропущенный управляющий спецификатор в определенном случае легко может привести к аварийному завершению работы системы. Даже простая строка символов *иногда* может привести к тяжким последствиям!

Нам еще осталось обсудить только большой ассемблерный блок внутри функции SpyHookInitializeEx(), расположенный между метками SpyHook2 и SpyHook9. Этот код обладает интересной особенностью: он никогда не выполняется при вызове SpyHookInitializeEx(). При входе в функцию код просто обходит весь ассемблерный блок, переходя на метку SpyHook9, расположенную перед разделом на языке C, где начинается инициализация массива aSpyHooks[]. Позже я покажу, как эти точки входа компонуются с таблицей SDT.

Одной из моих первоочередных задач при написании программы было создание кода, абсолютно не влияющего на выполнение других программ. Перехват вызовов операционной системы представляет опасность, поскольку никогда не известно, не зависит ли вызываемый код от некоторых неизвестных свойств контекста вызова. Теоретически достаточно соблюдать условия соглашения __stdcall, однако вероятность возникновения проблем при этом остается. Я решил предоставить первоначальному обработчику функций Native API практически точно такое же окружение, как если бы никаких перехватчиков не было бы вообще. Это значит, что функция Native API должна выполняться при первоначальном стеке аргументов, а значения всех регистров должны быть такими, как во время обращения к ней из вызывающей программы. Конечно, небольшая степень изменений должна быть допустима, так как иначе никакое наблюдение было бы невозможно. Наиболее существенное вмешательство в нашем случае — операции с адресом возврата в стеке. Если вы еще раз посмотрите на рис. 5.3, вы увидите, что при входе в функцию адрес возврата вызывающей функции расположен на вершине стека аргументов. Диспетчер перехвата в функции SpyHookInitializeEx() забирает этот адрес и помещает вместо него свой собственный адрес метки SpyHook6. Таким образом, первоначальный обработчик функции Native API после завершения работы перейдет на эту метку, позволяя диспетчеру перехвата просмотреть его аргументы и возвращаемые значения.

Перед тем как вызвать первоначальный обработчик, диспетчер устанавливает управляющий блок SPY_CALL (см. верхнюю часть листинга 5.3), содержащий необходимые диспетчеру впоследствии параметры. Некоторые параметры нужны для правильной записи вызова API, другие содержат информацию о вызывающей функции, чтобы диспетчер смог передать ей управление после записи данных наблюдения, как если бы ничего не произошло. Драйвер слежения содержит массив структур SPY_CALL в глобальном блоке DEVICE_CONTEXT, к которому можно обратиться через глобальную переменную gpDeviceContext в модуле w2k_spy.c Диспетчер перехвата ищет свободную структуру SPY_CALL, просматривая значение членов fInUse. Для загрузки и установки значения этого члена за одну операцию диспетчер использует инструкцию процессора XCHG. Это очень важно, потому что код выполняется в многопоточной среде, в которой обращения к глобальным данным на чтение и на запись должны быть защищены от возможного изменения этих данных другими потоками. Если свободная структура найдена, диспетчер сохраняет идентификатор потока вызывающей программы, полученный при помощи функции PsGetCurrentThreadId(), адрес структуры SPY_HOOK_ENTRY, связанной с текущей функцией API, адрес возврата вызывающей функции и весь стек аргументов. Число байт, занимаемых копируемыми аргументами, берется из массива KiArgumentTable, находящегося в системной таблице SDT. Если все структуры SPY_CALL заняты, исходный обработчик функции API вызывается без протоколирования.

Необходимость в массиве структур SPY_CALL возникает опять же из-за многопоточной среды Windows 2000. Довольно часто работа функции Native API откладывается, и управление получает другой поток, в течение выделенного ему кванта времени вызывающий другую функцию Native API. Это означает, что в диспетчер перехвата драйвера слежения в любое время может произойти повторный вход в

любой момент выполнения. Если диспетчер перехвата имел бы одну глобальную область для хранения SPY_CALL, она была бы затерта выполняющимся потоком перед тем, как ожидающий поток завершил бы с ней работу. В такой ситуации появление синего экрана смерти весьма вероятно. Чтобы получить лучшее представление об уровне вложенности вызовов функций Native API, я добавил в структуру DEVICE_CONTEXT члены dLevel и dMisses. При каждом повторном входе в диспетчер перехвата (то есть когда задействуется новая структура SPY_CALL) значение dLevel увеличивается на единицу. При достижении максимального уровня вложенности (то есть если не осталось свободных структур SPY_CALL) увеличивается значение dMisses, показывая, что для этого вызова запись не ведется. Мои наблюдения показали, что на практике уровень вложенности, как правило, не превышает четырех. Возможно, что при сильной загрузке повторный вход в функцию Native API будет происходить еще чаще, поэтому верхний предел вложенности я установил с запасом — он равен 256.

Перед вызовом исходного обработчика API диспетчер восстанавливает все регистры процессора, в том числе и EFLAGS, и осуществляет переход на точку входа функции. Это осуществляется непосредственно перед меткой SpyHook6 в листинге 5.3. На этот раз SpyHook6 находится на вершине стека, и за ней расположены аргументы вызывающей функции. После выхода из обработчика API управление передается обратно диспетчеру перехвата на метку SpyHook6. Выполняемый с этого момента код также спроектирован так, чтобы обеспечить минимально возможное вмешательство. Теперь основная цель — предоставить вызывающей функции по возможности точно такой же контекст вызова, какой был бы установлен первоначальным обработчиком функции API. Главная задача диспетчера сейчас — найти структуру SPY_CALL, в которой он сохранил информацию о текущем вызове API. Единственный надежный ключ поиска, которым обладает диспетчер, — это идентификатор потока вызывающей функции, который был сохранен в члене hThread структуры SPY_CALL. Поэтому диспетчер проходит в цикле по всему массиву структур SPY_CALL, проверяя значение идентификатора потока. Заметьте, что код не учитывает значение флага fInUse; в этом нет необходимости, поскольку у всех незанятых элементов значение hThread равно нулю, а это идентификатор потока бездействия (idle) системы. Цикл должен всегда завершаться до достижения конца массива, поскольку иначе диспетчер не сможет вернуть управление вызывающей программе, а это неисправимая ошибка. В этом случае у кода есть несколько вариантов действий, и он вызывает функцию KeBugCheck(), которая осуществляет управляемое завершение работы системы. Такая ситуация никогда не должна возникать, но если это все же произойдет, с системой должно было случиться что-то ужасное, поэтому завершение системы, скорее всего, будет наилучшим решением.

Если требуемая структура SPY_CALL будет найдена, работа диспетчера перехвата на этом почти заканчивается. В качестве последнего существенного действия он вызывает реализующую запись данных наблюдений функцию SpyHookProtocol(), передавая ей указатель на структуру SPY_CALL. Все, что нужно этой функции для создания отчета, хранится в этой структуре. После возврата из SpyHookProtocol() диспетчер освобождает элемент SPY_CALL, восстанавливает все регистры процессора и возвращает управление вызывающей программе.

Отчет перехвата функций API

Хорошая программа наблюдения за функциями API просматривает аргументы *после* вызова исходной функции, поскольку функция может возвращать дополнительные данные в буферах, переданных ей по ссылке. Поэтому основная записывающая функция SpyHookProtocol() вызывается в самом конце обработчика перехвата, непосредственно перед тем, как происходит возврат из функции API в вызывающую программу. Перед тем как обсуждать тайны ее реализации, изучите следующих два примера отчетов, чтобы знать заранее, как выглядит результат ее работы. На рис. 5.6 приведен снимок экрана при протоколировании файловых операций, произведенных при выполнении команды консоли dir c:\.

Сравните приведенные на рис. 5.6 записи в отчете с форматными строками из листинга 5.6. В примере 5.1 форматные строки функций NtOpenFile() и NtClose() соответствуют первой и четвертой строкам данных на рис. 5.6. Они очень похожи: для каждого управляющего спецификатора, предваряемого символом процента (см. табл. 5.2), в отчете выводится значение соответствующего параметра. Однако, как легко заметить, в отчете содержится и другая информация, не входящая в строки формата. Причину этого расхождения я объясню чуть позже.

Общий формат записи в отчете показан в примере 5.2. каждая запись состоит из фиксированного числа полей и разделителей между ними. Разделители позволяют программе легко осуществлять синтаксический разбор записей. Поля формируются на основе следующего набора простых правил:

- все численные величины приводятся в шестнадцатеричной нотации без лидирующих нулей и без обычного начального префикса «0x»;
- аргументы функции разделяются запятыми;
- аргументы-строки заключены в двойные кавычки;
- если аргумент-указатель равен NULL, его значение опускается;
- значения членов структуры разделены точками;

```
18:s0=NtOpenFile(+46C.18,n100001,o"\??\C:\",i0.1,n3,n4021)1BFEE5AE05B6710,278,2
19:s0=NtQueryInformationFile(!46C.18="\??\C:\",i0.6,p12E21C,n210,n9)1BFEE5AE05B6710,278,2
1A:s0=NtQueryVolumeInformationFile(!46C.18="\??\C:\",i0.12,p1321C8,n21C,n5)1BFEE5AE05B6710,278,2
1B:s0=NtClose(-46C.18="\??\C:\")1BFEE5AE05B6710,278,1
1C:s0=NtOpenFile(+46C.18,n100001,o"\??\C:\",i0.1,n3,n4021)1BFEE5AE05B6710,278,2
1D:s0=NtQueryInformationFile(!46C.18="\??\C:\",i0.6,p12E664,n210,n9)1BFEE5AE05B6710,278,2
1E:s0=NtQueryVolumeInformationFile(!46C.18="\??\C:\",i0.26,p1321C8,n220,n1)1BFEE5AE05B6710,278,2
1F:s0=NtClose(-46C.18="\??\C:\")1BFEE5AE05B6710,278,1
20:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n4021)1BFEE5AE05FFCA0,278,2
21:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.68,p12E994,n268,n3,bTRUE,u"*",bFALSE)1BFEE5AE05FFCA0,278,2
22:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.9FE,p139128,n1000,n3,bFALSE,u,bFALSE)1BFEE5AE05FFCA0,278,2
23:s80000006=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i80000006.0,p139128,n1000,n3,bFALSE,u,bFALSE)1BFEE5AE0661960,278,2
24:s0=NtClose(-46C.18="\??\c:\")1BFEE5AE0661960,278,1
25:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n800021)1BFEE5AE0661960,278,2
26:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\",i0.20,p12ED10,n20,n7)1BFEE5AE0661960,278,2
27:s0=NtClose(-46C.18="\??\c:\")1BFEE5AE0661960,278,1
```

Рис. 5.6. Пример отчета о наблюдении для команды dir c:\

Пример 5.1. Сравнение форматных строк с записями в отчете

```
"%s=NtOpenFile(%+.%n.%o.%1.%n.%n)"
18:s0=NtOpenFile(+46C.18.n100001.o"\??\C:\".i0.1.n3.n4021)BFEE5AE05B6710.27B.2
"%s=NtClose(%-)"
1B:s0=NtClose(-46 C1B="\??\C.\")BFEE5AE05B6710.27B.1
```

Пример 5.2. Общий формат записи в отчете

```
<#>:<status>=<function>(<arguments>)<time>.<thread>.<handles>
```

- соответствующие дескриптору имена объектов присоединяются к значению дескриптора после символа-разделителя «=»;
- метка дата/время задается в десятых долях микросекунды, истекших с 01-01-1601, — в базовом системном формате времени Windows 2000;
- идентификатор потока определяет уникальный числовой идентификатор потока, вызвавшего функцию API;
- счетчик дескрипторов отслеживает число дескрипторов, внесенных в данный момент в список дескрипторов драйвера слежения. На основе этого списка генерирующая отчет функция ищет имена объектов, соответствующих дескрипторам.

На рис. 5.7 показан еще один отчет о наблюдении за вызовами функций Native API при выполнении команды type c:\boot.ini в окне консоли. Ниже представлены пояснения к некоторым записям в отчете:

- в строке 0x31 функция NtCreateFile() вызывается для открытия файла \??\c:\boot.ini. (o"\??\c:\boot.ini\"). Функция вернула нулевой код состояния NTSTATUS (s0), то есть STATUS_SUCCESS, и выделила новый дескриптор файла со значением 0x18, принадлежащий процессу 0x46C (+46C.18). Следовательно, счетчик дескрипторов увеличился с единицы до двух;
- в строке 0x36 команда type считывает первые 512 байт (n200) из файла \??\c:\boot.ini в буфер по линейному адресу 0x0012F5B4 (p12F5B4), передавая дескриптор, полученный от функции NtCreateFile() (!46C.18="\??\c:\boot.ini"), функции NtReadFile(). Система успешно возвращает 512 байт (i0.200);

```
2D:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n4021)1BFEE5B075EE890,278,2
2E:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.6E,p12F4DC,n268,n3,bTRUE,u"boot.ini",bFALSE)1BFEE5B075EE890,278,2
2F:s80000006=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i80000006.0,p1389F0,n1000,n3,bFALSE,u,bFALSE)1BFEE5B07606FC0,278,2
30:s0=NtClose(-46C.18="\??\c:\")1BFEE5B07606FC0,278,1
31:s0=NtCreaterFile(+46C.18,n80100080,o"\??\c:\boot.ini",i0.1,I,n80,n3,n1n60,p,n0)1BFEE5B07606FC0,278,2
32:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E728,n8,n4)1BFEE5B07606FC0,278,2
33:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E778,n8,n4)1BFEE5B07606FC0,278,2
34:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.18,p12E758,n18,n5)1BFEE5B07606FC0,278,2
35:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07606FC0,278,2
36:s0=NtReadFile(!46C.18="\??\c:\boot.ini",p,p,p,i0.200,p12F5B4,n200,I,d)1BFEE5B07606FC0,278,2
37:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E780,n8,nE)1BREE5B07650550,278,2
38:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07650550,278,2
39:s0=NtReadFile(!46C.18="\??\c:\boot.ini",p,p,p,i0.48,p12F5B4,n200,I,d)1BFEE5B07650550,278,2
3A:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E780,n8,nE)1BFEE5B07650550,278,2
3B:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07650550,278,2
3C:s0=NtClose(-46C.18="\??\c:\boot.ini")1BFEE5B07650550,278,1
```

Рис. 5.7. Пример отчета о наблюдении для команды type c:\boot.ini

- в строке 0x39 дается указание считать еще один блок из файла (n200). На этот раз достигается конец файла, поэтому NtReadFile() возвращает только 75 байт (i0.4B). Отсюда можно вычислить размер моего файла boot.ini: 512 + 75 = 587 байт, что соответствует действительности;
- в строке 0x3C дескриптор файла \??\c:\boot.ini успешно освобождается функцией NtClose() (-46C.18="\??\c:boot.ini"), и счетчик дескрипторов уменьшается с двух до одного.

К этому моменту у вас уже должно сложиться представление о структуре отчета наблюдения за функциями API, что поможет вам понять все тонкости работы механизма создания отчетов. Как уже говорилось, главная функция, отвечающая за запись вызовов API, называется SpyHookProtocol() (листинг 5.7). Эта функция помещает записи отчета для каждого вызова функции API в кольцевой буфер на основе данных структуры SPY_CALL, полученной от диспетчера перехвата. Клиент драйвера слежения может читать этот отчет через вызовы IOCTL. Каждая запись представляет собой строку текста, которая завершается одним символом новой строки ('\n' в нотации C). Доступ к буферу отчета происходит в монопольном режиме, что реализовано при помощи объекта-мьютекса ядра KMUTEX kmProtocol, размещенного в глобальной структуре DEVICE_CONTEXT драйвера слежения. Функции SpyHookWait() и SpyHookRelease() запрашивают и освобождают этот мьютекс. Все обращения к буферу отчета должны предваряться вызовом SpyHookWait() и заканчиваться вызовом SpyHookRelease(), как показано в функции SpyHookProtocol().

Листинг 5.7. Главная функция записи данных перехвата SpyHookProtocol()

```
NTSTATUS SpyHookWait (void)
      {
      return MUTEX_WAIT (gpDeviceContext->kmProtocol);
      }

//  ------------------------------------------------------

LONG SpyHookRelease (void)
      {
      return MUTEX_RELEASE (gpDeviceContext->kmProtocol);
      }

//  ------------------------------------------------------

//  <#>:<status>=<function>(<arguments>)<time>,<thread>,<handles>

void SpyHookProtocol (PSPY_CALL psc)
      {
      LARGE_INTEGER liTime;
      PSPY_PROTOCOL psp = &gpDeviceContext->SpyProtocol;

      KeQuerySystemTime (&liTime);

      SpyHookWait ();

      if (SpyWriteFilter (psp,psc->pshe->pbFormat,
                          psc->adParameters,
                          psc->dParameters))
```

```
        {
        SpyWriteNumber(psp, 0, ++(psp->sh.dCalls)),          // <#>:
        SpyWriteChar       (psp, 0, ':');
                                                             // <status>=
        SpyWriteFormat  (psp, psc->pshe->pbFormat,           //  <function>
                              psc->adParameters),            //    (<arguments>)

        SpyWriteLarge  (psp, 0, &liTime);                 // <time>,
        SpyWriteChar   (psp, 0, ',');

        SpyWriteNumber(psp, 0, (DWORD) psc->hThread);     // <thread>,
        SpyWriteChar   (psp, 0, ',');

        SpyWriteNumber(psp, 0, psp->sh.dHandles);         // <handles>
        SpyWriteChar   (psp, 0, '\n');
        }
    SpyHookRelease ();
    return;
    }
```

Если сравнить основное тело функции SpyHookProtocol() из листинга 5.7 с общей схемой записи в отчете из примера 5.2, станет ясно, какой оператор создает каждое поле записи. Также станет ясно, почему строки отчета в листинге 5.6 не учитывают все данные записи — некоторые данные, не связанные с функциями, добавляются SpyHookProtocol() без форматных строк. Часть записи <status>=<function>(<arguments>) генерируется вызовом функции SpyWriteFormat() в самой середине SpyHookProtocol() на основе форматной строки, связанной с текущей функцией API, для которой ведется запись. Подробную информацию о реализации различных функций SpyWrite*() внутри драйвера слежения можно найти в исходных файлах w2k_spy.c и w2k_spy.h в каталоге \src\w2k_spy прилагающегося компакт-диска с примерами.

Заметьте, что этот код не очень надежный. Он был написан в 1997 г. для Windows NT и в то время работал как часы. Но после переноса программы в Windows 2000 время от времени возникали аварийные ситуации с появлением BSOD, когда перехватчики оставались установленными на длительное время. Хуже того, некоторые действия все время вызывали появление BSOD, например просмотр папки «My Computer» (Мой компьютер) в диалоговом окне File ▸ Open (Файл ▸ Открыть) моего любимого текстового редактора. Проанализировав многочисленные снимки памяти на момент сбоев, я обнаружил, что сбои были вызваны недопустимыми отличными от NULL значениями указателей, которые передавались некоторым функциям API. Как только драйвер слежения пытался обратиться по одному из таких указателей, система давала сбой. Наиболее вероятной причиной ошибки были указатели на структуры IO_STATUS_BLOCK и неверные указатели на строки внутри структур UNICODE_STRING и OBJECT_ATTRIBUTES. Я также обнаружил несколько структур UNICODE_STRING, члены Buffer которых не завершались нулем. Еще раз напоминаю вам: не следует полагать, что все строки типа UNICODE_STRING завершаются нулем. В сомнительном случае член Length всегда укажет нужно число байт, расположенных по адресу Buffer.

Чтобы исправить ошибку, я добавил проверку корректности указателей во все функции записи, которые должны были использовать клиентские указатели. Для этого я

применил обсужденную в главе 4 функцию SpyMemoryTestAddress(), которая проверяет, указывает ли линейный адрес на допустимую запись в таблице страниц (PTE). За подробностями обратитесь к листингам 4.22 и 4.24. В качестве альтернативы можно было бы добавить инструкции структурной обработки исключений (Structured Eception Handling, SEH) __try/__except.

Работа с дескрипторами

Важно отметить, что SpyHookProtocol() записывает вызов функции API, только если в операторе if функция SpyWriteFilter() возвращает TRUE. Этот прием помогает подавить отображение ненужной информации (мусора) в отчете о перехватах. Например, перемещение указателя мыши по экрану сопровождается рядом отвлекающих внимание вызовов NtReadFile(). Еще один источник мусора находит интересный аналог в физике: при измерении величины в условиях эксперимента сам факт измерения влияет на измеряемую величину и приводит к отклонению в результатах. То же самое происходит и при записи обращений к API. Заметьте, что функция NtDeviceIoControlFile() также включена в массив форматных строк в листинге 5.6. Однако для чтения отчета о перехвате функций API клиент драйвера слежения использует функции управления вводом-выводом. Это значит, что клиент найдет записи о своих собственных вызовах NtDeviceIoControlFile() в данных отчета. В зависимости от частоты обращений к IOCTL требуемые данные могут легко затеряться в этой лишней информации. Драйвер слежения решает эту проблему, запоминая идентификатор потока, установившего перехватчики API, чтобы игнорировать все вызовы API, генерируемые этим потоком.

Функция SpyWriteFilter() устраняет мусор, игнорируя все вызовы API, в которых участвуют дескрипторы, если вызов функции, создавшей дескриптор, не записывался. Если устройство слежения замечает, что дескриптор закрыт или иным образом возвращен системе, все следующие функции, использующие это значение дескриптора, также игнорируются. Такая техника подавляет запись всех вызовов API, в которых участвуют дескрипторы с долгим периодом действия, созданные системой или другими процессами перед началом ведения отчета о перехвате обращений к API. Конечно, фильтрацию можно включать и отключать от имени клиента средствами IOCTL. Удобство механизма фильтрации легко оценить при помощи клиентского приложения-примера, описанного далее (в этой главе). Работа, выполняемая этим простым «фильтром шума», просто поражает.

В листинге 5.6 дескрипторы генерируют функции NtCreateFile(), NtCreateKey(), NtOpenFile(), NtOpenKey(), NtOpenProcess() и NtopenThread(). В форматных строках всех этих функций содержится управляющий спецификатор %+, определенный в табл. 5.2 как «Дескриптор (зарегистрировать)». Закрывают дескриптор или делают его недопустимым функции NtClose() и NtDeleteKey(). У обеих функций в форматной строке присутствует спецификатор %-, обозначенный в табл. 5.2 как «Дескриптор (отменить регистрацию)». У других функций, просто использующих дескриптор, не создавая или освобождая его, в форматной строке находится управляющий форматом спецификатор %!. По существу, дескриптор — это просто число, которое уникально идентифицирует объект в контексте процесса. С технической сторо-

ны, дескриптор — это индекс записи в таблице дескрипторов, в которой содержатся свойства соответствующего объекта. Когда функция API использует новый дескриптор, клиент, как правило, должен передать структуру OBJECT_ATRIBUTES, которая, помимо прочего, содержит информацию об имени объекта, к которому требуется обратиться. Потом это имя нигде больше не требуется, поскольку система может найти все необходимые свойства объекта через дескриптор объекта и таблицу дескрипторов. Это не очень хорошо для пользователя драйвера слежения за вызовами API, поскольку ему приходится пробираться через бесчисленные записи отчета, содержащие бессмысленные числа вместо символьных имен. Поэтому мой драйвер слежения регистрирует все имена объектов вместе с соответствующими значениями дескрипторов и идентификаторами владеющих ими процессов, обновляя список с появлением нового дескриптора. Когда впоследствии появляется зарегистрированная пара дескриптор/процесс, функция записи извлекает из списка исходное символьное имя и добавляет его в отчет.

Дескриптор остается зарегистрированным все время, пока его явно не закроет функция API или он снова не появится в вызове API, создающем новый дескриптор. В Windows 2000 я часто замечал, что одно и то же значение дескриптора несколько раз возвращалось системой, хотя в отчете не отображался вызов, закрывавший бы перед этим дескриптор. Я не припомню такого в Windows NT 4.0. Зарегистрированный дескриптор, появляющийся снова с другими свойствами объекта, был, очевидно, как-то закрыт, поэтому необходимо отменить его регистрацию. Иначе каталог дескрипторов драйвера слежения со временем столкнется с переполнением.

Существенная часть этого механизма отслеживания дескрипторов — функция SpyWriterFilter(), вызываемая SpyHookProtocol() (см. листинг 5.7). Каждый вызов любой из перехватываемых функций API должен пройти через эту функцию. Ее реализация показана в листинге 5.8.

Листинг 5.8. Функция SpyWriterFilter() убирает нежелательные вызовы API из отчета

```
BOOL SpyWriteFilter (PSPY_PROTOCOL    psp
                     PBYTE            pbFormat,
                     PVOID            pParameters,
                     DWORD            dParameters)
    {
    PHANDLE                 phObject = NULL;
    HANDLE                  hObject  = NULL;
    POBJECT_ATTRIBUTES      poa                 = NULL,
    PDWORD                  pdNext;
    DWORD                   i, j;

    pdNext = pParameters;
    i = j  = 0;

    while (pbFormat [i])
        {
        while (pbFormat [i] && (pbFormat [i] != '%')) i++,

        if (pbFormat [i] && pbFormat [++i])
            {
            j++;
```

продолжение ⇩

Листинг 5.8 (*продолжение*)

```
            switch (pbFormat [J++])
                {
                case 'b':
                case 'a':
                case 'w':
                case 'u':
                case 'n':
                case 'l':
                case 's':
                case 'i':
                case 'c':
                case 'd':
                case 'p':
                    {
                    break;
                    }
                case 'o'
                    {
                    if (poa == NULL)
                        {
                        poa = (POBJECT_ATTRIBUTES) *pdNext;
                        }
                    break;
                    }
                case '+'
                    {
                    if (phObject == NULL)
                        {
                        phObject = (PHANDLE) *pdNext;
                        }
                    break;
                    }
                case '!':
                case '-'
                    {
                    if (hObject == NULL)
                        {
                        hObject = (HANDLE) *pdNext;
                        }
                    break;
                    }
            }       default
                    {
                    J--;
                    break;
                    }
                }
            pdNext++;
            }
        }
    return // число аргументов правильное
           (J == dParameters)
           &&
            // дескрипторы не используются
           (((phObject == NULL) && (hObject == NULL))
            ||
            // новый дескриптор, успешно зарегистрирован
```

```
        ((phObject != NULL) &&
         SpyHandleRegister (psp, PsGetCurrentProcessId (),
                            *phObject, OBJECT_NAME (poa)))
        ||
        // зарегистрированный дескриптор
        SpyHandleSlot (psp, PsGetCurrentProcessId (), hObject)
        ||
        // фильтр отключен
        (!gfSpyHookFilter));
}
```

По существу, функция SpyWriterFilter() просматривает форматную строку отчета, ища в ней спецификаторы %o (атрибуты объекта), %+ (новый дескриптор), %! (открытый дескриптор) и %- (закрытый дескриптор), и предпринимает специальные действия для определенных сочетаний:

- если никакие дескрипторы не задействованы, обращение к API всегда записывается. Это относится ко всем функциям API, форматные строки которых не содержат управляющих форматом спецификаторов), %+,%! и %-;
- если в форматную строку входит спецификатор %+, показывающий, что функция выделяет новый дескриптор, этот дескриптор регистрируется и связывается с именем первого элемента %o в форматной строке при помощи вспомогательной функции SpyHandleRegister(). Если ни одного такого элемента не существует, дескриптор регистрируется с пустой строкой. При успешной регистрации вызов записывается;
- если в форматной строке присутствуют спецификаторы %! или %+, вызванная функция использует или закрывает открытый дескриптор. В этом случае SpyWriteFilter() проверяет, зарегистрирован ли этот дескриптор, запрашивая его номер в списке через функцию SpyHandleSlot(). Если функция завершается успешно, вызов API записывается;
- во всех других случаях вызов записывается только тогда, когда отключен механизм фильтрации, что отражает логическая переменная gfSpyHookFilter.

Каталог дескрипторов входит в структуру SPY_PROTOCOL, включенную в глобальную структуру DEVICE_CONTEXT драйвера слежения w2k_sys.sys и определенную в листинге 5.9 вместе со своей вложенной структурой SPY_HEADER. После определений структур приведен исходный код четырех функций управления дескрипторами SpyHandleSlot(), SpyHandleName(), SpyHandleUnregister() и SpyHandleRegister(). Дескриптор регистрируется путем присоединения его значения в конец массива ahObjects[]. Одновременно идентификатор владеющего дескриптором процесса добавляется в массив ahProcesses[], имя объекта копируется в буфер awNames[], а начальное смещение имени записывается в массив adNames[]. Когда регистрация дескриптора отменяется, все эти действия отменяются, при этом все следующие за удаляемым элементы массива сдвигаются влево, чтобы ни в одном из массивов не осталось бы «дыр». Определения констант в начале листинга 5.9 задают размер каталога дескрипторов: в нем может быть записано до 4096 дескрипторов, предельный размер всех имен в совокупности ограничен 1 048 576 символами Unicode (2 Мбайт), и размер буфера отчета равен 1 Мбайт.

Листинг 5.9. Структуры и функции для управления дескрипторами

```
#define SPY_HANDLES       0x00001000     // максимальное число дескрипторов
#define SPY_NAME_BUFFER   0x00100000     // размер буфера имен объектов
#define SPY_DATA_BUFFER   0x00100000     // размер буфера данных отчета

//  -------------------------------------------------------

typedef struct _SPY_HEADER
      {
      LARGE_INTEGER      liStart;   // время запуска
      DWORD              dRead;     // индекс чтения данных
      DWORD              dWrite;    // индекс записи данных
      DWORD              dCalls;    // счетчик обращений к API
      DWORD              dHandles;  // счетчик дескрипторов
      DWORD              dName;     // индекс имени объекта
      }
      SPY_HEADER, *PSPY_HEADER, **PPSPY_HEADER;

#define SPY_HEADER_ sizeof (SPY_HEADER)

//  -------------------------------------------------------


typedef struct _SPY_PROTOCOL
      {
      SPY_HEADER     sh;                                 // заголовок отчета
      HANDLE         hProcesses    SPY_HANDLES];         // массив идентификаторов процессов
      HANDLE         hObjects      [SPY_HANDLES];        // массив дескрипторов
      DWORD          dNames        [SPY_HANDLES];        // смещения имен
      WORD           wNames        [SPY_NAME_BUFFER];    // строки имен
      BYTE           bData         [SPY_DATA_BUFFER];    // данные отчета
      }
      SPY_PROTOCOL, *PSPY_PROTOCOL, **PPSPY_PROTOCOL;

#define SPY_PROTOCOL_ sizeof (SPY_PROTOCOL)

//  -------------------------------------------------------

DWORD SpyHandleSlot (PSPY_PROTOCOL    psp,
                     HANDLE           hProcess,
                     HANDLE           hObject)
      {
      DWORD dSlot = 0;

      if (hObject != NULL)
          {
          while ((dSlot < psp->sh.dHandles)
                    &&
                    ((psp->ahProcesses [dSlot] != hProcess) ||
                     (psp->ahObjects   [dSlot] != hObject )))
              dSlot++;

          dSlot = (dSlot < psp->sh.dHandles ? dSlot+1 : 0);
          }
      return dSlot;
      }
```

```
//  ------------------------------------------------------

DWORD SpyHandleName (PSPY_PROTOCOL   psp,
                     HANDLE          hProcess,
                     HANDLE          hObject,
                     PWORD           pwName,
                     DWORD           dName)

     {
     WORD      w;
     DWORD     i;
     DWORD     dSlot = SpyHandleSlot (psp, hProcess, hObject);

     if ((pwName != NULL) && dName)
          {
          i = 0;

          if (dSlot)
               {
               while ((i+1 < dName) &&
                         (w = psp->awNames [psp->adNames [dSlot-1] + i]))
                    {
                    pwName [i++] = w;
                    }
               }
          pwName [i] = 0;
          }
     return dSlot;
     }

//  ------------------------------------------------------

DWORD SpyHandleUnregister (PSPY_PROTOCOL   psp,
                           HANDLE          hProcess,
                           HANDLE          hObject,
                           PWORD           pwName,
                           DWORD           dName)

     {
     DWORD i, j;
     DWORD dSlot = SpyHandleName (psp, hProcess, hObject,
                                  pwName, dName);
     if (dSlot)
          {
          if (dSlot == psp->sh.dHandles)
               {
               // удалить последний элемент массива имен
               psp->sh.dName = psp->adNames [dSlot-1]; .
               }
          else
               {
               i = psp->adNames [dSlot-1];
               j = psp->adNames [dSlot  ];

               // сдвинуть влево все остальные элементы массива имен
               while (j < psp->sh.dName)
                    {
```

продолжение

Листинг 5.9 (*продолжение*)

```
                    psp->awNames [i++] = psp->awNames [j++];
                    }
               j -= (psp->sh.dName = i);

               // сдвинуть влево все остальные элементы
               // массива дескрипторов и массива смещений имен
               for (i = dSlot; i < psp->sh.dHandles; i++)
                    {
                    psp->ahProcesses   [i-1] = psp->ahProcesses [i];
                    psp->ahObjects     [i-1] = psp->ahObjects   [i];
                    psp->adNames       [i-1] = psp->adNames     [i] - j;
                    }
               }
          psp->sh.dHandles--;
          }
     return dSlot;
     }

//  ------------------------------------------------------

DWORD SpyHandleRegister (PSPY_PROTOCOL     psp,
                         HANDLE            hProcess,
                         HANDLE            hObject,
                         PUNICODE_STRING   puName)
     {
     PWORD pwName;
     DWORD dName;
     DWORD i;
     DWORD dSlot = 0;

     if (hObject != NULL)
          {
          // отменить регистрацию старого дескриптора
          // с этим же значением
          SpyHandleUnregister (psp, hProcess, hObject, NULL, 0);

          if (psp->sh.dHandles == SPY_HANDLES)
               {
               // при переполнении отменить регистрацию
               // самого старого дескриптора
               SpyHandleUnregister (psp, psp->ahProcesses [0],
                                    psp->ahObjects [0], NULL, 0);
               }
          pwName = ((puName != NULL) &&
                    SpyMemoryTestAddress (puName)
                    ? puName->Buffer
                    : NULL);

          dName  = ((pwName != NULL) &&
                    SpyMemoryTestAddress (pwName)
                    ? puName->Length / WORD_
                    : 0);

          if (dName + 1 <= SPY_NAME_BUFFER - psp->sh.dName)
               {
               // присоединить объект в конец списка
               psp->ahProcesses   [psp->sh.dHandles] = hProcess;
               psp->ahObjects     [psp->sh.dHandles] = hObject;
```

```
            psp->adNames        [psp->sh.dHandles] = psp->sh.dName;

            for (i = 0; i < dName; i++)
                 {
                 psp->awNames [psp->sh.dName++] = pwName [i];
                 }
            psp->awNames [psp->sh.dName++] = 0;

            psp->sh.dHandles++;
            dSlot = psp->sh.dHandles;
            }
       }
  return dSlot;
  }
```

Управление перехватчиками API в пользовательском режиме

Выполняющийся в пользовательском режиме клиент драйвера слежения может управлять механизмом перехвата функций Native API и создаваемым отчетом при помощи функций IOCTL. Набор функций с именами вида SPY_IO_HOOK_* был упомянут в главе 4 при обсуждении функций слежения за памятью модуля w2k_spy.sys (см. листинг 4.7 и табл. 4.2).

Имеющая отношение к рассматриваемой теме часть табл. 4.2 повторяется ниже в табл. 5.3 Листинг 5.10 представляет собой выдержки из листинга 4.7, показывая механизм диспетчера функций управления перехватом. Каждая из этих функций будет рассмотрена в следующих разделах.

Таблица 5.3. Функции IOCTL управления перехватом, реализованные в драйвере слежения

Имя функции	ID	Код IOCTL	Описание
SPY_IO_HOOK_INFO	11	0x8000602C	Возвращает информацию о перехватчиках Native API
SPY_IO_HOOK_INSTALL	12	0x8000E030	Устанавливает перехватчики API
SPY_IO_HOOK_REMOVE	13	0x8000E034	Удаляет перехватчики API
SPY_IO_HOOK_PAUSE	14	0x8000E038	Приостанавливает/возобновляет протоколирование перехвата
SPY_IO_HOOK_FILTER	15	0x8000E03C	Включает/отключает фильтр журнала перехвата
SPY_IO_HOOK_RESET	16	0x8000E040	Очищает журнал перехвата
SPY_IO_HOOK_READ	17	0x80006044	Читает данные из журнала перехвата
SPY_IO_HOOK_WRITE	18	0x8000E048	Записывает данные в журнал перехвата

Листинг 5.10. Диспетчер команд перехвата драйвера слежения (частично)

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT   pDeviceContext,
                        DWORD             dCode,
                        PVOID             pInput,
                        DWORD             dInput,
                        PVOID             pOutput,
                        DWORD             dOutput,
                        PDWORD            pdInfo)
```

продолжение

Листинг 5.10 (*продолжение*)

```
    {
    SPY_MEMORY_BLOCK    smb;
    SPY_PAGE_ENTRY      spe;
    SPY_CALL_INPUT      sci;
    PHYSICAL_ADDRESS    pa;
    DWORD               dValue, dCount;
    BOOL                fReset, fPause, fFilter, fLine;
    PVOID               pAddress;
    PBYTE               pbName;
    HANDLE              hObject;
    NTSTATUS            ns = STATUS_INVALID_PARAMETER;

    MUTEX_WAIT  (pDeviceContext->kmDispatch);

    *pdInfo = 0;

    switch (dCode)
        {
//  ============================================
// Лишние функции IDCTL опущены (см. листинг 4.7)
//  ============================================
        case SPY_ID_HOOK_INFO:
            {
            ns = SpyOutputHookInfo (pOutput, dOutput, pdInfo);
            break;
            }
        case SPY_IO_HOOK_INSTALL:
            {
            if (((ns = SpyInputBool (&fReset,
                                     pInput, dInput))
                 == STATUS_SUCCESS)
                &&
                ((ns = SpyHookInstall (fReset, &dCount))
                == STATUS_SUCCESS))
                {
                ns = SpyOutputDword (dCount,
                                     pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_REMOVE:
            {
            if (((ns = SpyInputBool (&fReset,
                                     pInput, dInput))
                 == STATUS_SUCCESS)
                &&
                ((ns = SpyHookRemove (fReset, &dCount))
                 == STATUS_SUCCESS))
                {
                ns = SpyOutputDword (dCount,
                                     pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_PAUSE:
```

```
                {
                if ((ns = SpyInputBool (&fPause,
                                        pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    fPause = SpyHookPause (fPause);

                    ns = SpyOutputBool (fPause,
                                        pOutput, dOutput, pdInfo);
                    }
                break;
                }
            case SPY_IO_HOOK_FILTER:
                {
                if ((ns = SpyInputBool (&fFilter,
                                        pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    fFilter = SpyHookFilter (fFilter);

                    ns = SpyOutputBool (fFilter,
                                        pOutput, dOutput, pdInfo);
                    }
                break;
                }
            case SPY_IO_HOOK_RESET:
                {
                SpyHookReset ();
                ns = STATUS_SUCCESS;
                break;
                }
            case SPY_IO_HOOK_READ:
                {
                if ((ns = SpyInputBool (&fLine,
                                        pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputHookRead (fLine,
                                            pOutput, dOutput, pdInfo);
                    }
                break;
                }
            case SPY_IO_HOOK_WRITE:
                {
                SpyHookWrite (pInput, dInput);
                ns = STATUS_SUCCESS;
                break;
                }
//  ============================================
// Лишние функции IOCTL опущены (см. листинг 4.7)
//  ============================================
        }
    MUTEX_RELEASE  (pDeviceContext->kmDispatch);
    return ns;
    }
```

Функция IOCTL SPY_IO_HOOK_INFO

Функция IOCTL SPY_IO_HOOK_INFO заполняет структуру SPY_HOOK_INFO информацией о текущем состоянии механизма перехвата и информацией из таблицы SDT системы. В эту структуру входят различные другие структуры (или ссылки на них), представленные ранее:

- SERVICE_DESCRIPTOR_TABLE определена в листинге 5.1;
- SPY_CALL и SPY_HOOK_ENTRY определены в листинге 5.2;
- SPY_HEADER и SPY_PROTOCOL определены в листинге 5.9.

Листинг 5.11. Определение структуры SPY_HOOK_INFO

```
typedef struct _SPY_HOOK_INFO
      {
      SPY_HEADER                  sh.
      PSPY_CALL                   psc:
      PSPY_PRDTOCDL               psp:
      PSERVICE_DESCRIPTOR_TABLE   psdt.
      SERVICE_DESCRIPTOR_TABLE    sdt.
      DWORD                       ServiceLimit;
      NTPROC                      ServiceTable   [SDT_SYMBOLS_MAX];
      BYTE                        ArgumentTable  [SDT_SYMBOLS_MAX];
      SPY_HDDK_ENTRY              SpyHooks       [SDT_SYMBOLS_MAX];
      }
      SPY_HOOK_INFO, *PSPY_HOOK_INFO, **PPSPY_HOOK_INFO,

#define SPY_HOOK_INFO_ sizeof (SPY_HOOK_INFO)
```

Будьте внимательны при получении значений членов этой структуры. Некоторые из них являются указателями на память режима ядра, недоступную из пользовательского режима. Всегда можно воспользоваться функцией драйвера слежения SPY_IO_MEMORY_DATA для изучения содержимого соответствующих блоков памяти.

Функция IOCTL SPY_IO_HOOK_INSTALL

Функция IOCTL SPY_IO_HOOK_INSTALL модифицирует таблицу системных вызовов ntoskrnl.exe внутри SDT системы, устанавливая точки входа из глобального массива aSpyHooks[]. Этот массив заполняется данными функциями SpyHookInitialize() (листинг 5.5) и SpyHookInitializeEx() (листинг 5.3) при инициализации драйвера. Каждый элемент массива aSpyHooks[] содержит точку входа перехватчика и адрес соответствующей форматной строки, если она существует. Функция SpyDispatcher() для установки перехватчиков вызывает вспомогательную функцию SpyHookInstall() из листинга 5.12, которая при выполении задачи, в свою очередь, обращается к функции SpyHookExchange() из того же листинга.

Листинг 5.12. Модификация таблицы системных вызовов Native API

```
DWORD SpyHookExchange (void)
      {
      PNTPRDC   ServiceTable:
      BOOL      fPause.
```

```
    DWORD     i;
    DWORD     n = 0.

    fPause         = SpyHookPause (TRUE).
    ServiceTable   =  KeServiceDescriptorTable->ntoskrnl ServiceTable.

    for.(i = 0; i < SDT_SYMBOLS_MAX, i++)
        {
        if (aSpyHooks [i].pbFormat != NULL)
            {
            aSpyHooks [i].Handler = (NTPROC)
                InterlockedExchange ((PLONG) ServiceTable+i.
                                     (LONG)  aSpyHooks [i] Handler).
            n++;
            }
        }
    gfSpyHookState = !gfSpyHookState.
    SpyHookPause (fPause);
    return n.
    }

//  ------------------------------------------------------

NTSTATUS SpyHookInstall (BOOL     fReset.
                         PDWORD  pdCount)
    {
    DWORD          n  = 0;
    NTSTATUS  ns  = STATUS_INVALID_DEVICE_STATE.

    if (!gfSpyHookState)
        {
        ghSpyHookThread = PsGetCurrentThreadId ().

        n = SpyHookExchange ().
        if (fReset) SpyHookReset ().

        ns = STATUS_SUCCESS.
        }
    *pdCount = n;
    return ns
    }
```

Функция SpyHookExchange() принимает участие и при установке, и при удалении перехватчиков, она просто меняет местами записи из таблицы системных вызовов Native API и элементы массива aSpyHooks[]. Поэтому если дважды вызвать эту функцию, таблица вызовов и массив вернутся в свое исходное состояние. Функция SpyHookExchange() в цикле проходит по массиву aSpyHooks[], ища элементы, содержащие указатель на форматную строку. Если такая строка существует, это означает, что следует осуществлять наблюдение за этой функцией. В этом случае указатель на функцию API в таблице системных вызовов и член Handler структуры — элемента массива aSpyHooks[] меняются местами при помощи функции InterlockedExchange() из модуля ntoskrnl.exe, которая гарантирует, что никакой другой поток не сможет повлиять на эту операцию. На все время изменения таблицы системных вызовов механизм протоколирования временно приостанавливается

Функция SpyHookInstall() просто образует оболочку над SpyHookExchange(), выполняя при этом некторые дополнительные действия:

- таблица системных вызовов не модифицируется, если глобальный флаг gfSpyHookState показывает, что перехватчики уже установлены;
- идентификатор потока вызывающей функции записывается в глобальную переменную ghSpyHookThread(). Диспетчер перехвата из функции SpyHookInitializeEx() при помощи этой информации подавляет все вызовы API, приходящие из этого потока. Если этого не сделать, отчет о перехвате будет заполнен ненужной и мешающей информацией о взаимодействии драйвера слежения и его клиента пользовательского режима;
- отчет сбрасывается по запросу клиента, то есть все содержимое буфера отбрасывается и каталог дескрипторов снова инициализируется.

Функция SPY_IO_HOOK_INSTALL принимает от вызывающей функции логический параметр. Если он равен TRUE, отчет сбрасывается после установки перехватчиков. Этот параметр используется чаще всего. Передача в качестве параметра FALSE продолжает отчет с того места, на котором закончился предыдущий сеанс перехвата. Функция возвращает количество замененных записей в таблице системных вызовов. В Windows 2000 SPY_IO_HOOK_INSTALL дает значение 44, которое равно числу элементов в массиве форматных строк apbSdtFormats[] в листинге 5.6. В Windows NT 4.0 устанавливаются только 42 перехватчика, потому что функции Native API NtNotifyChangeMultipleKeys() и NtQueryOpenSubKeys() в ней не поддерживаются.

Функция IOCTL SPY_IO_HOOK_REMOVE

Функция IOCTL SPY_IO_HOOK_REMOVE похожа на SPY_IO_HOOK_INSTALL, поскольку выполняет главным образом обратные ей действия. Аргументы ввода-вывода IOCTL этих функций одинаковы. Однако вспомогательная функция SpyHookRemove(), вызываемая из SpyDispatcher(), в некоторых важных моментах отличается от SpyHookInstall(), как показывает сравнение листингов 5.12 и 5.13:

- вызов игнорируется, если глобальный флаг gfSpyHookState показывает, что в данный момент никаких перехватчиков не установлено;
- после восстановления таблицы системных вызовов в исходное состояние идентификатор потока клиента, установившего обработчики, очищается путем присваивания глобальной переменной ghSpyHookThread нулевого значения;
- наиболее важная дополнительная особенность функции — цикл do/while в середине листинга 5.13. В этом цикле SpyHookRemove() проверяет, не работает ли в данный момент диспетчер перехвата с другими потоками, проверяя члены fInUse всех структур SPY_CALL, входящих в глобальную структуру DEVICE_CONTEXT. Это необходимо, поскольку клиент может попытаться выгрузить драйвер слежения сразу после установки перехватчиков. Если это произойдет в то время, когда обращения к API из какого-либо другого потока все еще обрабатываются диспетчером перехвата, система возбудит исключение с появлением BSOD. Проверки проводятся с интервалом в 100 миллисекунд, чтобы дать другим потокам время на выход из драйвера слежения.

Листинг 5.13. Восстановление таблицы системных вызовов Native API

```
NTSTATUS SpyHookRemove (BOOL     fReset,
                        PDWORD  pdCount)
      {
      LARGE_INTEGER       liDelay;
      BOOL                fInUse;
      DWORD               i;
      DWORD               n    = 0;
      NTSTATUS            ns   = STATUS_INVALID_DEVICE_STATE;

      if (gfSpyHookState)
           {
           n = SpyHookExchange ();
           if (fReset) SpyHookReset ();

           do  {
                for (i = 0; i < SPY_CALLS; i++)
                     {
                     if (fInUse = gpDeviceContext->SpyCalls [i].fInUse)
                          break;
                     }
                liDelay.QuadPart = -1000000;
                KeDelayExecutionThread (KernelMode, FALSE, &liDelay);
                }
           while (fInUse);

           ghSpyHookThread = 0;

           ns = STATUS_SUCCESS;
           }
      *pdCount = n;
      return ns;
      }
```

Обратите внимание: даже если ни один флаг fInUse не установлен, последняя задержка в 100 миллисекунд все равно добавляется. Такая предосторожность требуется, поскольку в диспетчере перехвата есть крошечная дыра в безопасности. Ее создает код между инструкцией, в которой сбрасывается флаг текущего элемента-структуры SPY_CALL, и инструкцией RET, возвращающей управление из диспетчера в вызывающую программу (см. листинг 5.2, код между метками SpyHoook8 и SpyHook9). Если все флаги fInUse установлены в FALSE, все равно остается небольшая вероятность, что некоторые потоки были приостановлены сразу перед выполнением инструкции RET. Задержка удаления перехватчиков еще на 100 миллисекунд дает всем потокам время на то, чтобы покинуть этот критический участок кода.

Функция IOCTL SPY_IO_HOOK_PAUSE

Функция IOCTL SPY_IO_HOOK_PAUSE, показанная в листинге 5.14, позволяет клиенту на время отключать и снова включать функцию записи перехватов. Для этого функция устанавливает глобальную переменную gfSpyHookPause в логическое значение, переданное клиентом, и возвращает предыдущее значение этой переменной при помощи функции API InterlockedExchange() из модуля ntoskrnl.exe. По умолчанию протоколирование включено, то есть переменная gfSpyHookPause установлена в FALSE.

Важно отметить, что SPY_IO_HOOK_PAUSE работает полностью независимо от функций SPY_IO_HOOK_INSTALL и SPY_IO_HOOK_REMOVE. Если запись отчета будет приоста-

новлена во время установки перехватчиков, они будут установлены, но диспетчер перехвата никак не будет затрагивать все вызовы API. Можно также отключить протоколирование перед установкой перехватчиков, если вы не хотите, чтобы запись в отчет включалась автоматически после того, как SPY_IO_HOOK_INSTALL модифицирует таблицу системных вызовов. Отметьте, что при возобновлении записи отчет автоматически начинается заново.

Листинг 5.14. Включение и отключение записи отчета

```
BOOL SpyHookPause (BODL fPause)
      {
      BOOL fPause1 = (BOOL)
                     InterlockedExchange ((PLONG) &gfSpyHookPause,
                                          (LONG)  fPause);
      if (!fPause) SpyHookReset ();
      return fPause1;
      }
```

Функция IOCTL SPY_IO_HOOK_FILTER

Функция IOCTL SPY_IO_HOOK_FILTER оперирует с глобальным флагом gfSpyHookFilter, как показано в листинге 5.15. Флаг устанавливается в переданное клиентом значение, а его предыдущее значение возвращается в вызывающую функцию. По умолчанию флаг установлен в FALSE, то есть фильтр отключен.

Листинг 5.15. Включение и отключение фильтрации протокола

```
BOOL SpyHookFilter (BOOL fFilter)
      {
      return (BOOL) InterlockedExchange ((PLONG) &gfSpyHookFilter,
                                         ( LONG) fFilter);
      }
```

Переменная gfSpyHookFilter должна быть вам уже знакома из определения функции SpyWriteFilter() в листинге 5.8. Если флаг gfSpyHookFilter равен TRUE, эта функция помогает SpyHookProtocol() (листинг 5.7) отбросить все вызовы API, в которых задействованы дескрипторы, не отмеченные ранее драйвером слежения.

Функция IOCTL SPY_IO_HOOK_RESET

Функция IOCTL SPY_IO_HOOK_RESET устанавливает механизм записи в исходное состояние, очищая буфер данных и список дескрипторов. Функция SpyHookReset(), вызываемая SpyDispatcher(), является просто оболочкой функции SpyWriteReset(). Обе функции показаны в листинге 5.16. SpyHookReset(), кроме того, добавляет реализацию монопольного доступа к буферу отчета при помощи функций SpyHookWait() и SpyHookRelease() (см. листинг 5.7).

Функция IOCTL SPY_IO_HOOK_READ

Данные отчета записываются в буфер abData[], принадлежащий глобальной структуре SPY_PROTOCOL, показанной в листинге 5.9. Этот массив байт трактуется как кольцевой буфер, то есть для него поддерживается пара указателей для доступа на

запись и на чтение. Как только один из указателей достигнет конца буфера, он переустанавливается в его начало. Указатель чтения всегда пытается догнать указатель записи, в пустом буфере оба указателя указывают на одно и то же место.

SPY_IO_HOOK_READ, безусловно, самая важная функция управления перехватом в драйвере слежения. Она читает произвольное количество данных из буфера данных отчета и перемещает соответственно указатель чтения. При включенной записи ее следует вызывать как можно чаще, чтобы избежать переполнения буферов. В листинге 5.17 показан набор функций, обрабатывающих этот запрос IOCTL. Основную работу выполняют функции SpyReadData() и SpyReadLine(). Они отличаются друг от друга тем, что первая возвращает по возможности запрашиваемое количество данных, а вторая извлекает данные только одной строки. Построчный режим может быть очень удобным, если клиентское приложение должно как-то фильтровать данные. Функции, вызывающие SPI_IO_HOOK_READ, передают ей логическое значение, определяющее блочный (FALSE) или построчный (TRUE) режим.

Листинг 5.16. Переустановка отчета

```
void SpyWriteReset (PSPY_PROTOCOL psp)
      {
      KeQuerySystemTime  (&psp->sh.liStart).

      psp->sh.dRead      = 0.
      psp->sh.dWrite     = 0.
      psp->sh.dCalls     = 0;
      psp->sh.dHandles   = 0;
      psp->sh.dName      = 0;
      return;
      }

//  ------------------------------------------------------

void SpyHookReset (void)
      {
      SpyHookWait        ().
      SpyWriteReset      (&gpDeviceContext->SpyProtocol).
      SpyHookRelease     ().
      return;
      }
```

Листинг 5.17. Чтение из буфера отчета

```
DWORD SpyReadData (PSPY_PROTOCOL   psp.
                   PBYTE           pbData.
                   DWORD           dData)
      {
      DWORD i = psp->sh.dRead.
      DWORD n = 0.

      while ((n < dData) && (i != psp->sh.dWrite))
            {
            pbData [n++] = psp->abData [i++];
            if (i == SPY_DATA_BUFFER) i = 0;
            }
      psp->sh.dRead = i.
      return n;
      }
```

продолжение

Листинг 5.17 (*продолжение*)

```
//  ------------------------------------------------------

DWORD SpyReadLine (PSPY_PROTOCOL  psp,
                   PBYTE          pbData,
                   DWORD          dData)
      {
      BYTE  b = 0;
      DWORD i = psp->sh.dRead;
      DWORD n = 0;

      while ((b != '\n') && (i != psp->sh.dWrite))
           {
           b = psp->abData [i++];
           if (i == SPY_DATA_BUFFER) i = 0;
           if (n < dData) pbData [n++] = b;
           }
      if (b == '\n')
           {
           // удалить текущую строку из буфера
           psp->sh.dRead = i;
           }
      else
           {
           // не возвращать данные,
           // пока не будет считана вся строка
           n = 0;
           }
      if (n)
           {
           pbData [n-1] = 0;
           }
      else
           {
           if (dData) pbData [0] = 0;
           }
      return n;
      }

//  ------------------------------------------------------

DWORD SpyHookRead (PBYTE  pbData,
                   DWORD  dData,
                   BOOL   fLine)
      {
      DWORD n = 0;

      SpyHookWait ();

      n = (fLine ? SpyReadLine : SpyReadData)
              (&gpDeviceContext->SpyProtocol, pbData, dData);

      SpyHookRelease ();
      return n;
      }
```

```
//  ---------------------------------------------------------

NTSTATUS SpyOutputHookRead (BOOL     fLine,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD   dInfo)
      {
      *pdInfo = SpyHookRead (pOutput, dOutput, fLine);
      return STATUS_SUCCESS;
      }
```

Функции SpyOutpurHookRead() и SpyHookRead() тривиальны. SpyHookRead() добавляет обычную реализацию монопольного доступа при помощи мьютекса и выбирает одну из функций SyReadLine() и SpyReadData(), а SpyOutpurHookRead() затем обрабатывает ее результаты так, как этого требует механизм IOCTL.

Функция IOCTL SPY_IO_HOOK_WRITE

Функция IOCTL SPY_IO_HOOK_WRITE позволяет клиенту записывать данные в буфер отчета. Приложению может понадобиться такая возможность для добавления в отчет разделителей или дополнительной информации о состоянии. Реализация функции показана в листинге 5.18. SpyHookWrite() — еще одна функция-оболочка, в которую добавлена реализация монопольного доступа посредством мьютекса. Она вызывает функцию SpyWriteData(), основной генератор данных отчета в драйвере слежения. Все вспомогательные функции SpyWrite*() в листинге 5.7 (то есть SpyWriteFormat(), SpyWriteNumber(), SpyWriteChar() и SpyWriteLarge()) построены на ее основе.

Листинг 5.18. Запись в буфер отчета

```
DWORD SpyWriteData (PSPY_PROTOCOL   psp,
                    PBYTE           pbData,
                    DWORD           dData)
      {
      BYTE  b;
      DWORD i = psp->sh.dRead;
      DWORD j = psp->sh.dWrite;
      DWORD n = 0;

      while (n < dData)
           {
           psp->abData [j++] = pbData [n++];
           if (j == SPY_DATA_BUFFER) j = 0;

           if (j == i)
                {
                // удалить из буфера первую строку
                do  {
                     b = psp->abData [i++];
                     if (i == SPY_DATA_BUFFER) i = 0;
                     }
                while ((b != '\n') && (i != j));

                // если в буфере только одна строка, удалить ее первую половину
```

продолжение ⇩

Листинг 5.18 (*продолжение*)

```
                if ((i == j) &&
                    ((i += (SPY_DATA_BUFFER / 2)) >= SPY_DATA_BUFFER))
                    {
                    i -= SPY_DATA_BUFFER
                    }
                }
            }
        psp->sh dRead  = i,
        psp->sh dWrite = j,
        return n,
        }

//  ------------------------------------------------------

DWORD SpyHookWrite (PBYTE  pbData,
                    DWORD  dData)
        {
        DWORD n = 0;

        SpyHookWait (),

        n = SpyWriteData
                (&gpDeviceContext->SpyProtocol, pbData, dData);

        SpyHookRelease (),
        return n,
        }
```

Обратите внимание на то, как SpyWriteData() обрабатывает ситуации переполнения. Если указатель чтения продвигается медленно, указатель записи может его обогнать. В этой ситуации возможны два способа действий:

1. Доступ на запись запрещается, пока указатель чтения не продвинется вперед.
2. Данные в буфере отбрасываются для освобождения места.

Драйвер слежения выбирает вторую возможность. В случае переполнения отбрасывается вся строка отчета в текущей позиции указателя чтения и указатель продвигается к следующей строке. Если в буфере содержится только одна строка (что очень маловероятно), будет отброшена только первая половина строки. В листинге 5.18 обрабатывающий эти две ситуации код помечен соответствующими комментариями.

Пример программы чтения отчета

Чтобы вам было легче создавать собственные клиентские приложения перехвата API, я создал очень простую программу-пример, которая читает буфер отчета о перехвате и отображает его в окне консоли. Вызов функций паузы, фильтрации и переустановки буфера осуществляется нажатием клавиш P, F и R на клавиатуре, а выходные данные можно фильтровать в соответствии с определенными пользователем шаблонами имен функций. Приложение называется «SBS Windows 2000 API Hook Viewer», его исходный код находится на прилагающемся к книге компакт-диске в каталоге \src\w2k_hook.

Управление драйвером слежения

Для удобства в приложении w2k_hook.exe используются несколько простых оболочек для различных функций IOCTL SPY_IO_HOOK_*, приведенных в листинге 5.19. Применение этих вспомогательных функций существенно облегчает чтение кода и снижает вероятность ошибок при передаче параметров при разработке клиентского приложения для драйвера слежения.

Листинг 5.19. Вспомогательные функции для работы с IOCTL

```
BOOL WINAPI SpyIoControl (HANDLE   hDevice,
                          DWORD    dCode,
                          PVOID    pInput,
                          DWORD    dInput
                          PVOID    pOutput,
                          DWORD    dOutput)
      {
      DWORD dInfo = 0;

      return DeviceIoControl (hDevice,  dCode,
                              pInput,   dInput,
                              pOutput,  dOutput,
                              &dInfo,   NULL)
               &&
               (dInfo == dOutput);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyVersionInfo (HANDLE               hDevice,
                            PSPY_VERSION_INFO    psvi)
      {
      return SpyIoControl (hDevice, SPY_IO_VERSION_INFO,
                                    NULL, 0,
                                    psvi, SPY_VERSION_INFO_);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyHookInfo (HANDLE          hDevice,
                         PSPY_HOOK_INFO  pshi)
      {
      return SpyIoControl (hDevice, SPY_IO_HOOK_INFO,
                                    NULL, 0,
                                    pshi, SPY_HOOK_INFO_);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyHookInstall (HANDLE   hDevice,
                            BOOL     fReset,
                            PDWDRD   pdCount)
      {
      return SpyIoControl (hDevice, SPY_IO_HOOK_INSTALL,
                                    &fReset, BOOL_,
                                    pdCount, DWORD_);
      }
```

продолжение

Листинг 5.19 (*продолжение*)

```
//  ------------------------------------------------------

BOOL WINAPI SpyHookRemove (HANDLE  hDevice,
                           BOOL    fReset,
                           PDWORD  pdCount)
      {
      return SpyIoControl (hDevice,  SPY_IO_HOOK_REMOVE,
                           &fReset,  BOOL_,
                           pdCount,  DWORD_);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyHookPause (HANDLE  hDevice,
                          BOOL    fPause,
                          PBOOL   pfPause)
      {
      return SpyIoControl (hDevice,  SPY_ID_HOOK_PAUSE,
                           &fPause,  BOOL_,
                           pfPause,  BOOL_);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyHookFilter (HANDLE  hDevice,
                           BOOL    fFilter,
                           PBOOL   pfFilter)
      {
      return SpyIoControl (hDevice,  SPY_IO_HOOK_FILTER,
                           &fFilter, BOOL_,
                           pfFilter, BOOL_);
      }

//  ------------------------------------------------------

BOOL WINAPI SpyHookReset (HANDLE hDevice)
      {
      return SpyIoControl (hDevice, SPY_IO_HOOK_RESET,
                           NULL.0,
                           NULL, 0);
      }

//  ------------------------------------------------------

DWORD WINAPI SpyHookRead (HANDLE  hDevice,
                          BOOL    fLine,
                          PBYTE   pbData,
                          DWORD   dData)
      {
      DWORD dInfo;

      if (!DeviceIoControl (hDevice, SPY_IO_HOOK_READ,
                            &fLine,  BOOL_,
                            pbData,  dData,
                            &dInfo,  NULL))
```

```
        {
        dInfo = 0;
        }
    return dInfo;
    }

//  ----------------------------------------------------

BOOL WINAPI SpyHookWrite (HANDLE  hDevice,
                          PBYTE   pbData)
    {
    return SpyIoControl (hDevice,   SPY_IO_HOOK_WRITE,
                         pbData,    lstrlenA (pbData),
                         NULL,      0);
    }
```

Перед тем как обращаться к функциям из листинга 5.19, необходимо загрузить и запустить драйвер слежения. Это делается точно так же, как и в главе 4 в случае приложения просмотра памяти w2k_mem.exe. В листинге 5.20 показана основная функция приложения, Execute(), которая загружает и выгружает драйвер слежения, открывает и закрывает дескриптор устройства и взаимодействует с драйвером через IOCTL. Если сравнить листинги 5.20 и 4.29, сразу будет заметна схожесть в начальной и конечной частях. Различается только середина программы, в которой находится специфический для приложения код.

Листинг 5.20. Каркас основного приложения

```
void WINAPI Execute (PPWORD  ppwFilters,
                     DWORD   dFilters)
    {
    SPY_VERSION_INFO    svi;
    SPY_HOOK_INFO       shi;
    DWORD               dCount, i, j, k, n;
    BOOL                fPause, fFilter, fRepeat;
    BYTE                abData [HOOK_MAX_DATA];
    WORD                awData [HOOK_MAX_DATA];
    WORD                awPath [MAX_PATH]  = L"?";
    SC_HANDLE           hControl           = NULL;
    HANDLE              hDevice            = INVALID_HANDLE_VALUE;

    _printf (L"\r\nLoading \"%s\" (%s) ...\r\n",
                  awSpyDisplay, awSpyDevice);

    if (w2kFilePath (NULL, awSpyFile, awPath, MAX_PATH))
        {
        _printf (L"Driver: \"%s\"\r\n",
                      awPath);

        hControl = w2kServiceLoad (awSpyDevice,
                                   awSpyDisplay,
                                   awPath, TRUE);
        }
    if (hControl != NULL)
        {
        _printf (L"Opening \"%s\" ...\r\n",
                  awSpyPath);
```

продолжение

Листинг 5.20 (*продолжение*)

```
        hDevice = CreateFile (awSpyPath,
                              GENERIC_READ        | GENERIC_WRITE,
                              FILE_SHARE_READ     | FILE_SHARE_WRITE,
                              NULL, OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL, NULL);
        }
    else
    {
_printf (L"Unable to load the spy device driver.\r\n");
        }
    if (hDevice != INVALID_HANDLE_VALUE)
        {
        if (SpyVersionInfo (hDevice, &svi))
            {
            _printf (L"\r\n"
                     L"%s V%lu.%02lu ready\r\n",
                     svi.awName,
                     svi.dVersion / 100, svi.dVersion % 100);
            }
        if (SpyHookInfo (hDevice, &shi))
            {
            _printf (L"\r\n"
            L"API hook parameters:        0x%08lX\r\n"
            L"SPY_PROTOCOL structure:     0x%08lX\r\n"
            L"SPY_PROTOCOL data buffer:   0x%08lX\r\n"
            L"KeServiceDescriptorTable:   0x%08lX\r\n"
            L"KiServiceTable:             0x%08lX\r\n"
            L"KiArgumentTable:            0x%08lX\r\n"
            L"Service table size:         0x%lX (%lu)\r\n",
                     shi.psc,
                     shi.psp,
                     shi.psp->abData,
                     shi.psdt,
                     shi.sdt.ntoskrnl.ServiceTable,
                     shi.sdt.ntoskrnl.ArgumentTable,
                     shi.ServiceLimit, shi.ServiceLimit);
            }
        SpyHookPause  (hDevice, TRUE, &fPause );
        fPause  = FALSE;
        SpyHookFilter (hDevice, TRUE, &fFilter);
        fFilter = FALSE;

        if (SpyHookInstall (hDevice, TRUE, &dCount))
            {
            _printf (L"\r\n"
                     L"Installed %lu API hooks\r\n",
                     dCount);
            }
        _printf (L"\r\n"
                 L"Protocol control keys:\r\n"
                 L"\r\n"
                 L"P    -  pause  ON/off\r\n"
                 L"F    -  filter ON/off\r\n"
                 L"R    -  reset protocol\r\n"
                 L"ESC  -  exit\r\n"
                 L"\r\n");
```

```
for (fRepeat = TRUE; fRepeat;)
    {
    if (n = SpyHookRead (hDevice, TRUE,
                         abData, HOOK_MAX_DATA))
        {
        if (abData [0] == '-')
            {
            n = 0;
            }
        else
            {
            i = 0;
            while (abData [i] && (abData [i++]
                    != '='));

            j = i;
            while (abData [j] && (abData [j] != '('))
                j++;

            k = 0;
            while (i < j)
                awData [k++] = abData [i++];

            awData [k] = 0;

            for (i = 0; i < dFilters; i++)
                {
                if (PatternMatcher (ppwFilters [i],
                                    awData))
                    {
                    n = 0;
                    break;
                    }
                }
            }
        if (!n) _printf (L"%hs\r\n", abData);
        Sleep (0);
        }
    else
        {
        Sleep (HOOK_IOCTL_DELAY);
        }
    switch (KeyboardData ())
        {
        case 'P':
            {
            SpyHookPause (hDevice, fPause, &fPause);
            SpyHookWrite (hDevice, (fPause
                                    ? abPauseOff
                                    : abPauseOn));
            break;
            }
        case 'F':
            {
            SpyHookFilter (hDevice, fFilter,
                           &fFilter);
```

продолжение

Листинг 5.20 (*продолжение*)

```
                    SpyHookWrite  (hDevice, (fFilter
                                             ? abFilterOff
                                             : abFilterOn));
                    break;
                    }
               case 'R':
                    {
                    SpyHookReset (hDevice);
                    SpyHookWrite (hDevice, abReset);
                    break;
                    }
               case VK_ESCAPE:
                    {
                    _printf (L"%hs\r\n", abExit);
                    fRepeat = FALSE;
                    break;
                    }
               }
          }
     if (SpyHookRemove (hDevice, FALSE, &dCount))
          {
          _printf (L"\r\n"
                   L"Removed %lu API hooks\r\n",
                   dCount);
          }
     _printf (L"\r\nClosing the spy device ...\r\n");
     CloseHandle (hDevice);
     }
else
     {
     _printf (L"Unable to open the spy device.\r\n");
     }
if ((hControl != NULL) && gfSpyUnload)
     {
     _printf (L"Unloading the spy device ...\r\n");
     w2kServiceUnload (awSpyDevice, hControl);
     }
return;
}
```

Заметьте, что функция Execute() в листинге 5.20 запрашивает доступ GENERIC_READ и GENERIC_WRITE в вызове функции CreateFile(), а функция из листинга 4.29 использует доступ только GENERIC_READ. Различие вызвано кодами IOCTL, которые использует приложение. В то время как программа просмотра памяти из главы 4 всегда использует функции только для чтения, программа перехвата вызовов API, обсуждаемая здесь, обращается к функциям, изменяющим системные данные, и поэтому требует дескриптора устройства с доступом на запись. Если изучить коды IOCTL в третьем столбце табл. 5.3, можно заметить, что у большинства из них четвертую позицию справа занимает шестнадцатеричная цифра E, а у функций SPY_IO_HOOK_INFO и SPY_IO_HOOK_READ в этом месте стоит цифра 6. В соответствии со схемой на рис. 4.6 в главе 4 это означает, что паре функций управления перехватом SPY_IO_HOOK_INFO и SPY_IO_HOOK_READ требуется дескриптор устройства с доступом на чтение, а остальным функциям необходимы права на чтение/запись. Проектиров-

щик драйвера устройства должен решить, какая комбинация прав на чтение и на запись требуется для обработки запросов ввода-вывода этим устройством. Модификация таблицы системных вызовов, безусловно, требует записи данных, поэтому будет правильно предоставить клиенту дескриптор с доступом на запись.

Большая часть остального кода в листинге 5.20 должна быть понятна без объяснений. Некоторые моменты все же стоит отметить:

- функция SPY_IO_HOOK работает в строковом режиме, что показывает второй аргумент при вызове SpyHookRead() в начале большого цикла for;
- пользователь приложения может задать в командной строке ряд шаблонов строк с использованием символов замещения (wildcards) '*' и '?'. Шаблоны последовательно сравниваются с именем функции в каждой строке отчета при помощи вспомогательной функции PatternMatcher(), показанной в листинге 5.21. Если шаблону не удовлетворяет ни одно имя, строка не выводится. Чтобы просмотреть полный неотфильтрованный отчет, нужно выполнить команду w2k_hook *;
- после обработки строки отчета приложение приостанавливает свою работу на 10 миллисекунд (HOOK_IOCTL_DELAY), перед тем как снова запросить данные у драйвера слежения. Это значительно сокращает загрузку процессора в периоды редкого обращения к Native API;
- в главном цикле происходит также опрос клавиатуры. Все клавиши, кроме P, F, R и Esc игнорируются. P (pause) управляет режимом паузы (по умолчанию он включен), F (filter) включает и отключает фильтрацию по дескриптору (по умолчанию она включена), R (reset) начинает отчет заново и Esc завершает работу приложения;
- если нажата одна из клавиш P, F, R или Esc, в буфер отчета при помощи функции SPY_IO_HOOK_WRITE записывается линия-разделитель. Строка отмечает смену режима после введенной команды. Запись разделителя в буфер лучше, чем вывод его напрямую в окно консоли, поскольку смена режима может проявиться на экране с некоторой задержкой. Например, если нажата клавиша P для прекращения вывода на экран, приложение будет продолжать выводить данные, пока не считает их все из буфера отчета. Созданный при выполнении команды P разделитель будет присоединен после последней записи, поэтому он появится в правильном месте;
- аналогично приложению w2k_mem.exe из главы 4, w2k_hook.exe выгружает драйвер слежения только при установленном глобальном флаге gfSpyUnload. По умолчанию этот флаг *не* установлен — по причинам, изложенным в главе 4.

Листинг 5.21. Простая функция проверки на совпадение с шаблоном

```
BOOL WINAPI PatternMatcher (PWORD pwFilter,
                            PWORD pwData)
    {
    DWORD i, j;

    i = j = 0;
    while (pwFilter [i] && pwData [j])
```

продолжение

Листинг 5.21 (*продолжение*)

```
        {
        if (pwFilter [i] != '?')
            {
            if (pwFilter [i] == '*')
                {
                i++.
                if ((pwFilter [i] != '*') && (pwFilter [i]
                     != '?'))
                    {
                    if (pwFilter [i])
                        {
                        while (pwData [j] &&
                                 (!PatternMatcher (pwFilter + i.
                                                   pwData + j)))
                            {
                            j++.
                            }
                        }
                    return (pwData [j])
                    }
                }
            if ((WORD) CharUpperW ((PWORD) (pwFilter [i]))
                !=
                 (WORD) CharUpperW ((PWORD) (pwData     [j])))
                {
                return FALSE.
                }
            }
        i++.
        j++.
        }
    if (pwFilter [i] == '*') i++.
    return !(pwFilter [i] || pwData [j]).
    }
```

Примеры, показанные на рис. 5.6 и 5.7, были созданы при запуске программы w2k_hook.exe с шаблонами имен *file и ntclose, заданными в командной строке. Такая команда отфильтровывает все вызовы функций управления файлами и функцию NtClose(). Важно знать, что шаблоны имен применяются к данным отчета *после* того, как он был создан, а основанный на регистрации дескрипторов фильтр «мусора» в драйвере слежения влияет на отчет *до* записи. Если исключить записи отчета, указав шаблоны имен в командной строке w2k_hook.exe, это ни в коей мере не повлияет на функцию создания данных отчета. Будет происходить только отбор записей отчета после извлечения из буфера.

Основные тезисы и проблемы

Модифицированный здесь механизм перехвата API, первоначально разработанный Руссиновичем и Когсвеллом, остроумен и элегантен. Основные преимущества данного подхода следующие:

- установка и удаление перехватчика в таблицу системных вызовов реализуется при помощи простой операции обмена указателями;

- после того как перехватчик установлен, он получает вызовы Native API от всех выполняющихся в системе процессов, даже от новых, запущенных после установки перехватчика;
- поскольку драйвер перехвата работает в режиме ядра, он обладает полным доступом ко всем системным ресурсам. Разрешено даже выполнять привилегированные инструкции процессора.

Во время разработки драйвера слежения я столкнулся с проблемами в следующих областях:

- программа драйвера перехвата должна проектироваться и быть написана с чрезвычайной осторожностью. Поскольку все вызовы уровня Native API будут осуществляться в контексте различных потоков приложений, драйвер должен быть так же стабилен, как само ядро операционной системы. Малейшее упущение немедленно приведет к аварийной остановке системы;
- записывается только небольшая часть трафика API ядра. Например, вызовы API, исходящие из других модулей режима ядра, не проходят через шлюз INT 2Eh системы и поэтому не появляются в отчете о перехватах. Кроме того, многие важные функции, экспортируемые модулями ntdll.dll и ntoskrnl.exe, не являются частью Native API, поэтому их нельзя перехватить через таблицу системных вызовов.

Неполный охват API, конечно, более существенное ограничение, чем требование стабильности. В любом случае удивительно, как много полезной информации можно получить о внутренней организации приложения путем отслеживания его обращений к Native API. Например, я смог подробно изучить действия протокола NCP (Netware Core Protocol), производимые программой-редиректором NetWare фирмы Microsoft, просто наблюдая его вызовы функции NtFsControlFile(). Таким образом, данный подход к наблюдению API безусловно наиболее профессиональный из всех возможных на сегодня альтернатив для Windows 2000.

Вызов функций API ядра из пользовательского режима

В главе 2 я объяснял, как приложения пользовательского режима в Windows 2000 могут обращаться к подмножеству функций API ядра системы, Native API, при помощи механизма шлюза прерываний. В главах 4 и 5 для выполнения дополнительных задач, не разрешенных в режиме пользователя, задействуется механизм управления вводом-выводом устройства (Device I/O Control, IOCTL). Оба средства весьма эффективны, но подумайте о перспективах, открывающихся при существовании возможности вызвать практически любую функцию режима ядра, как если бы она находилась в обычной DLL пользовательского режима. Обычно это считается невозможным, но я в этой главе покажу, как это сделать при помощи некоторых нетривиальных программных приемов. Здесь на помощь снова придет технология IOCTL, решая на первый взгляд неразрешимую проблему. Эта глава кардинально меняет привычные представления, демонстрируя способ создания универсального моста между пользовательским режимом и режимом ядра, позволяющего приложению Win32 обращаться к функциям API ядра просто как к функциям интерфейса Win32 API. Более того, при помощи файлов идентификаторов инструментов отладки Windows 2000 приложение сможет вызывать внутренние функции ядра, недоступные даже драйверам режима ядра. Такой «интерфейс обращения к ядру» работает в фоновом режиме практически полностью незаметно для вызывающего приложения.

Общий интерфейс обращения к ядру

В главе 4 для вызова определенных функций API ядра от имени программы пользовательского режима мы использовали драйвер режима ядра. Например, функция SPY_IO_PHYSICAL из драйвера слежения w2k_spy.sys является просто оболочкой функ-

ции диспетчера памяти MmGetPhysicalAddress(). Другим примером может служить функция SPY_IO_HANDLE_INFO, созданная на основе функций диспетчера объектов ObReferenceObjectByHandle() и ObDereferenceObject(). Хотя эта техника прекрасно работает, было бы довольно утомительно и неэффективно разрабатывать собственную функции IOCTL для каждой функции API ядра, к которой требуется доступ из пользовательского режима. Поэтому я разработал и включил в драйвер слежения w2k_spy.sys универсальную функцию IOCTL, способную вызвать любую функцию режима ядра по ее символьному имени или точке входа, сопровождаемой списком аргументов. Кажется, что это потребует серьезных усилий, но вы удивитесь, насколько простым окажется код. Есть только одна трудность — нам снова придется интенсивно работать со встроенным ассемблером.

Разработка шлюза для обращений к режиму ядра

Когда выполняющейся в пользовательском режиме программе требуется вызвать функцию режима ядра, ей необходимо решить две проблемы. Во-первых, она должна как-то преодолеть барьер между режимом пользователя и режимом ядра, и, во-вторых, программа должна осуществлять обмен данными. Для подмножества функций интерфейса Native API эту работу выполняет ntdll.dll, осуществляя смену режима при помощи шлюза прерываний и используя регистры процессора для передачи функции ядра указателя на стек аргументов вызывающей функции и возврата результата. Для функций ядра, не входящих в состав Native API, в операционной системе нет подобного шлюза, следовательно, нужно создать наш собственный. Первая часть проблемы решается легко: драйвер слежения w2k_spy.sys, описанный в главе 4 и расширенный в главе 5, много раз туда и обратно переходит границу между режимом пользователя и режимом ядра во время операций IOCTL. И поскольку при помощи IOCTL можно передавать блоки данных в обоих направлениях, проблему передачи данных также можно считать решенной. В конечном итоге вся задача сводится к следующей простой последовательности действий:

1. Приложение пользовательского режима отправляет IOCTL запрос, передавая информацию о вызываемой функции ядра и указатель на стек ее аргументов.
2. Драйвер режима ядра обрабатывает запрос, копирует аргументы в собственный стек, вызывает функцию и передает результат обратно в выходном буфере IOCTL.
3. Вызывающая функция принимает результаты операции IOCTL и работает дальше точно так же, как и после обычного вызова функции из DLL.

Основная сложность этого алгоритма в том, что модуль режима ядра должен уметь работать с различными форматами данных и соглашениями о передаче параметров. Вот список ситуаций, с которыми должен справляться драйвер:

- размер стека аргументов зависит от конкретной функции ядра. Поскольку было бы непрактично хранить в драйвере информацию обо всех возможных функциях, вызывающая программа должна передавать размер стека аргументов;

- в функциях API ядра Windows 2000 используются три соглашения о вызовах: __stdcall, __cdscl и __fastcall, в которых способы передачи параметров весьма различны По требованиям соглашений __stdcall и __cdecl все аргументы должны передаваться через стек, а цель соглашения __fastcall — уменьшить накладные расходы при работе со стеком, для чего два первых параметра передаются через регистры процессора ECX и EDX. С другой стороны, у соглашений __stdcall и __fastcall одинаковый способ удаления аргументов из стека — это возлагается на вызываемую функцию, а в соглашении __cdecl, напротив, стек должна очищать вызывающая функция. Проблему очистки стека можно легко решить, сохраняя значение указателя стека до вызова и восстанавливая его после возврата, однако самостоятельно драйвер никак не сможет распознать соглашение __fastcall. Поэтому вызывающая функция должна при каждом вызове указывать, применяется ли соглашение __fastcall, чтобы драйвер смог в случае необходимости подготовить регистры ECX и EDX;
- функции ядра Windows 2000 возвращают значения различных размеров, от нуля бит до 64. Возвращаемое значение передается обратно через 64-битную пару регистров EDX:EAX. Данные записываются начиная с младших разрядов, то есть, к примеру, если функция возвращает 16-битное значение типа SHORT, оно будет помещено в регистр AX (состоящий из AL и AH). Старшая половина регистра EAX и все содержимое регистра EDX будут не определены. Поскольку драйверу ничего неизвестно о возвращаемых функцией ядра данных, необходимо рассчитывать на худший случай — 64 бита, так как иначе данные результата могут быть потеряны;
- приложение может передать недопустимые аргументы. В пользовательском режиме в этом, как правило, нет ничего страшного. В худшем случае процесс приложения аварийно завершится с сообщением об ошибке. Изредка такая ошибка приводит к повреждению системных данных, что требует перезагрузки и восстановления системы. В режиме ядра самая распространенная программная ошибка — неверное значение указателя — почти наверняка приведет к появлению синего экрана смерти, что может сопровождаться потерей пользовательских данных. Эту проблему во многом решает механизм структурной обработки исключений (Structured Exception Handling, SEH) операционной системы.

Теперь давайте посмотрим, как наш драйвер слежения работает с аргументами, возвращаемыми значениями и свойствами функций. В листинге 6.1 показаны используемые механизмом IOCTL структуры для входных и выходных данных, SPY_CALL_INPUT и SPY_CALL_OUTPUT. Последняя структура весьма проста — она состоит из объединения ULARGE_INTEGER, представляющего в Windows 2000 64-битное значение одновременно как 64-битное целое и как две 32-битные части. Подробнее о ULARGE_INTEGER см. листинг 2.3 в главе 2.

Листинг 6.1. Определение структур SPY_CALL_INPUT и SPY_CALL_OUTPUT

```
typedef struct _SPY_CALL_INPUT
    {
    BOOL   fFastCall.
    DWORD  dArgumentBytes;
    PVOID  pArguments;
```

```
    PBYTE pbSymbol;
    PVOID pEntryPoint.
    }
    SPY_CALL_INPUT. *PSPY_CALL_INPUT. **PPSPY_CALL_INPUT.

#define SPY_CALL_INPUT_ sizeof (SPY_CALL_INPUT)

//  ------------------------------------------------------

typedef struct _SPY_CALL_OUTPUT
    {
    ULARGE_INTEGER ullResult.
    }
      SPY_CALL_OUTPUT.
     *PSPY_CALL_OUTPUT.
    **PPSPY_CALL_OUTPUT:

#define SPY_CALL_OUTPUT_ sizeof (SPY_CALL_OUTPUT)
```

Структура SPY_CALL_INPUT нуждается в пояснении. Назначение флага fFastCall должно быть понятно сразу — он сообщает драйверу, что вызываемая функция ядра применяет соглашение __fastcall, поэтому первые два аргумента (если они есть) необходимо передать не через стек, а через регистры процессора. Член структуры dArgumentBytes задает число байт в стеке аргументов, а pArguments указывает на вершину этого стека. Оставшиеся члены структуры, pbSymbol и pEntryPoint, взаимно исключают друг друга и определяют функцию ядра либо через ее идентификатор (pbSymbol), либо через простой адрес точки входа (pEntryPoint). Другой член всегда должен быть установлен в NULL. Если оба значения не равны NULL, pbSymbol имеет предпочтение перед pEntryPoint. Вызов функции по имени длиннее на один шаг, поскольку при этом требуется определить точку входа для указанного идентификатора. Если ее можно получить, вход в функцию будет осуществлен по этому адресу. Этот шаг будет пропущен, если точка входа будет передана в качестве параметра

Однако не так-то просто найти линейный адрес, соответствующий экспортируемому модулем режима ядра идентификатору. Функции Win32 GetModuleHandle() и GetProcessAddress(), прекрасно работающие для всех компонентов подсистемы Win32, не могут распознавать модули и драйверы режима ядра. Реализация этой части программы-примера далась нелегко, все подробности будут описаны в следующем разделе этой (главы). На данный момент будем считать, что мы каким-то образом получили корректную точку входа. В листинге 6.2 показана функция SpyCall(), играющая основную роль в моем интерфейсе вызовов ядра. Как видите, она почти полностью написана на ассемблере. Никогда не хочется прибегать к ассемблеру в программе на C, но некоторые вещи просто невозможно сделать на чистом C. В нашем случае проблема в том, что функции SpyCall() необходим полный контроль над стеком и регистрами процессора, поэтому ей нужно блокировать действия компилятора и оптимизатора C, которые используют стек и регистры на свое усмотрение.

Перед тем как перейти к подробному рассмотрению листинга 6.2, я хотел бы описать еще одну особенность функции SpyCall(), из-за которой код выглядит несколько менее понятным. Как объяснялось в главе 2, системные модули Windows 2000 экс-

портируют некоторые переменные, например NtBuildNumber и KeServiceDescriptorTable, по имени. Формат Portable Executable (PE) систем Windows 2000/NT/9x предоставляет универсальный механизм привязки идентификаторов к адресам независимо от того, указывает ли адрес на код или данные. Поэтому модуль Windows 2000 может по своему желанию назначать экспортируемые идентификаторы своим глобальным переменным. Компоновка этих идентификаторов с клиентским модулем происходит динамически, аналогично компоновке идентификаторов функций, и клиентский модуль может использовать эти переменные точно так же, как если бы они располагались в его собственном разделе глобальных данных. Мой интерфейс вызовов ядра был бы неполон, если бы не мог обрабатывать этот вид идентификаторов, поэтому я решил, что отрицательные значения члена dArgumentBytes структуры SPY_CALL_INPUT будут означать, что данные следует копировать из точки входа функции ядра, а не получать в виде возвращаемого значения. Допустимы значения в диапазоне от –1 до –9, где –1 означает, что в буфер SPY_CALL_OUTPUT копируется сам адрес точки входа. Дополнения до единицы остальных возможных значений показывают количество байт, которые копируются, начиная с точки входа. То есть если значение равно –2, копируется один байт типа BYTE или CHAR, –3 соответствует 16-битному типу WORD или SHORT, –5 — 32-битному DWORD или LONG и –9 соответствует 64-битному DWORDLONG или LONGLONG. Вы можете спросить, зачем может потребоваться копировать саму точку входа. Дело в том, что некоторые идентификаторы ядра, например KeServiceDescriptorTable, указывают на структуры, размер которых превышает границу возвращаемого значения в 64 бита, поэтому лучше будет передать один указатель, чем обрезать значение до 64 бит.

Листинг 6.2. Ключевая функция интерфейса вызовов ядра

```
void SpyCall (PSPY_CALL_INPUT psci,
              PSPY_CALL_OUTPUT   psco)
    {
    PVOID pStack;

    __asm
        {
        pushfd
        pushad
        xor     eax, eax
        mov     ebx, psco                   ; получить блок
                                            ; выходных параметров
        lea     edi, [ebx.ulResult]         ; получить буфер
                                            ; результата
        mov     [edi  ], eax                ; очистить буфер результата
                                            ; (младшую часть)
        mov     [edi+4], eax                ; очистить буфер результата
                                            ; (старшую часть)
        mov     ebx, psci                   ; получить блок
                                            ; входных парметров
        mov     ecx, [ebx dArgumentBytes]
        cmp     ecx, -9                     ; вызов или копирование?
        jb      SpyCall2
        mov     esi, [ebx pEntryPoint]      ; получить точку входа
        not     ecx                         ; получить число байт
        jecxz   SpyCall1                    ; 0 -> сохранить точку входа
```

```
        rep     movsb                           ; скопировать данные из точки входа
        jmp     SpyCall5
SpyCall1:
        mov     [edi], esi                      ; сохранить точку входа
        jmp     SpyCall5
SpyCall2:
        mov     esi, [ebx pArguments]
        cmp     [ebx fFastCall], eax            ; соглашение
                                                ; __fastcall?
        jz      SpyCall3
        cmp     ecx, 4                          ; существует первый аргумент?
        jb      SpyCall3
        mov     eax, [esi]                      ; eax = первый аргумент
        add     esi, 4                          ; удалить аргумент из списка
        sub     ecx, 4
        cmp     ecx, 4                          ; существует второй аргумент?
        jb      SpyCall3
        mov     edx, [esi]                      ; edx = второй аргумент
        add     esi, 4                          ; удалить аргумент из списка
        sub     ecx, 4
SpyCall3:
        mov     pStack, esp                     ; сохранить указатель стека
        jecxz   SpyCall4                        ; нет (больше) аргументов
        sub     esp, ecx                        ; скопировать стек аргументов
        mov     edi, esp
        shr     ecx, 2
        rep     movsd
SpyCall4:
        mov     ecx, eax                        ; загрузить первый
                                                ; аргумент _fastcall
        call    [ebx.pEntryPoint]               ; вызвать функцию
        mov     esp, pStack                     ; восстановить указатель
                                                ; стека
        mov     ebx, psco                       ; получить блок выходных
                                                ; параметров
        mov     [ebx.u1iResult.LowPart ], eax   ; сохранить
                                                ; результат (младшая часть)
        mov     [ebx.u1iResult.HighPart], edx   ; сохранить
                                                ; результат (старшая часть)
SpyCall5:
        popad
        popfd
        }
    return:
    }
```

Если принять во внимание особый случай доступа к экспортируемым переменным, листинг 6.2 не должен вызывать затруднений. Сначала очищается 64-битный буфер результата, чтобы неиспользуемые биты всегда были равны нулю. Затем член dArgumentBytes входной структуры сравнивается со значением –9, чтобы определить вид действия — вызов функции или копирование данных. Обработчик вызова функции начинается с метки SpyCall2. После этого регистр ESI устанавливается в значение, указывающее на вершину стека аргументов, определяемое значением pArguments. Далее проверяется соглашение о вызовах. Если задано соглашение __fastcall и в стеке есть по крайней мере одно 32-битное значение, SpyCall()

удаляет его из стека и временно сохраняет в EAX. Если есть еще одно 32-битное значение, оно также удаляется из стека и сохраняется в EDX, остальные аргументы остаются в стеке. Тем временем мы подошли к метке SpyCall3. В этот момент текущий (исходный) адрес вершины стека драйвера слежения сохраняется в локальной переменной pStack. После этого в стек драйвера копируется стек аргументов (за вычетом двух удаленных аргументов в случае соглашения __fastcsll) при помощи быстрой инструкции процессора i386 REP MOVSD. Заметьте, что флаг, определяющий направление продвижения инструкции MOVSD в памяти (вверх или вниз) можно считать сброшенным по умолчанию, то есть регистры ESI и EDI увеличиваются после каждого шага копирования. Перед выполнением инструкции CALL осталось только скопировать первый аргумент __fastcall из предварительного места хранения в регистр ECX. SpyCall() слепо копирует значение регистра EAX в ECX, поскольку эта операция не оказывает негативного воздействия в случае соглашений о вызовах __stdcall или __cdecl. Инструкция MOV ECX, EAX настолько быстра, что намного эффективней выполнить ее лишний раз, чем осуществлять проверку значения члена fFastCall, для того чтобы обойти эту инструкцию в случае других соглашений.

После возврата из функции SpyCall() переустанавливает указатель стека в исходное значение, ранее сохраненное в переменной pStack. Таким образом учитывается различие в способе очистки стека между соглашениями _stdcall и __fatscall и соглашением __cdecl. По соглашению __cdecl, при возврате из функции в вызывающую подпрограмму регистр ESP указывает на вершину стека аргументов, а функция, работающая по соглашению __stdcall или __fastcall, сбрасывает регистр, устанавливая его в значение исходного адреса перед вызовом. Если после возврата из функции принудительно присвоить регистру ESP значение ранее сохраненного исходного адреса, стек всегда будет очищен правильно независимо от используемого функцией соглашения. В последних двух ассемблерных строках функции SpyCall() значение, возвращенное функцией API ядра в регистрах EDX:EAX, записывается в структуру SPY_CALL_OUTPUT, которая, в свою очередь, будет передана в функцию, вызвавшую SpyCall(). Не делается попыток определить правильный размер результата. В этом нет необходимости, поскольку вызывающая функция точно знает нужное количество бит. Скопированные лишние биты будут проигнорированы.

По поводу функции в листинге 6.2 необходимо заметить следующее: ее код никак не предохранен от неверных аргументов, не проверяется даже корректность самого указателя стека. В режиме ядра создание такого кода равносильно игре с огнем. Но, с другой стороны, как драйверу слежения проверить все аргументы? 32-битное значение в стеке может быть значением счетчика, массивом битовых полей или указателем. Смысл передаваемых аргументов знает только исходная функция режима пользователя и функция API ядра. Функция SpyCall() просто передает данные, ничего не зная о их типе. Если добавить в нее проверку типов аргументов, необходимо было бы переписать довольно большие фрагменты операционной системы. К счастью, в Windows 2000 существует простой способ выхода из этой ситуации: структурная обработка исключений (SEH).

SEH представляет удобную для использования среду, позволяющую программе перехватывать исключения, которые в ином случае привели бы к сбою системы.

Исключение — это аварийная ситуация, которая заставляет процессор остановить выполнение текущих действий. Как правило, исключения возникают при чтении или записи по линейному адресу, которому не соответствует страница в физической памяти или в файле подкачки, записи данных в сегмент кода, попытке выполнять инструкции в сегменте данных или делении числа на ноль. Не все исключения критичны. Например, исключение, генерируемое при обращении к памяти, которая выгружена в файл подкачки, может быть обработано системой путем загрузки требуемой страницы в физическую память. Все-таки большая часть исключений приводит к сбою, поскольку операционная система не знает, как ей исправить ошибку; в таких ситуациях она просто завершает свою работу. Такое поведение системы может показаться чрезмерно жестким, но иногда лучше немедленно остановить приближающуюся катастрофу, пока ситуация не стала еще хуже. Механизм SEH дает программе, вызвавшей исключение, возможность исправить ситуацию. При помощи специальных определенных Microsoft конструкций языка C __try/__except для обработки исключения можно задать любой набор инструкций. Если исключение приведет систему в критическое состояние, активизируется собственный обработчик программы, позволяя программисту указать более разумную реакцию, чем просто отображение синего экрана (смерти).

Конечно, техникой SEH можно воспользоваться и для решения проблемы корректности параметров в нашем драйвере слежения. В листинге 6.3 показана оболочка, помещающая функцию SpyCall() в оболочку SEH. Защищенный код заключен в фигурные скобки инструкции __try. Естественно, защищается не только инструкция SpyCall(), но и весь связанный с ней код, выполняющийся в контексте вызова. При возбуждении исключения начинает выполняться код внутри инструкции __except, как определяет выражение-фильтр EXCEPTION_EXECUTE_HANDLER. Обработчик исключения в листинге 6.3 тривиален: он просто заставляет SpyCallEx() вернуть код состояния STATUS_ACCESS_VIOLATION вместо STATUS_SUCCESS, что, в свою очередь, приведет к ошибке при вызове DeviceIoControl() на стороне пользовательского режима. Синего экрана смерти не будет, единственная проблема в том, что результаты вызываемой функции будут не определены, но вызывающая программа должна быть к этому готова в любом случае.

Листинг 6.3. Применение структурной обработки исключений (SEH) в интерфейсе вызовов ядра

```
NTSTATUS SpyCallEx (PSPY_CALL_INPUT   psci,
                    PSPY_CALL_OUTPUT  psco)
    {
    NTSTATUS ns = STATUS_SUCCESS.

    __try
        {
        SpyCall (psci. psco).
        }
    __except  (EXCEPTION_EXECUTE_HANDLER)
        {
        ns = STATUS_ACCESS_VIOLATION.
        }
    return ns.
    }
```

Хотя механизм SEH отлавливает самые распространенные ошибки параметров, не следует ожидать от него защиты от любого мусора, который клиентское приложение может передать функции API ядра. Задание недопустимых значений некоторых аргументов функций приводят к сбою системы, не возбуждая исключения. Например, функция копирования строк легко может затереть жизненно важные части памяти системы, если указатель на буфер результирующей строки указывает на неверный адрес. Такого рода ошибки могут никак не проявляться в течение длительного времени, пока система вдруг неожиданно не даст сбой, когда работающая программа со временем обратится к измененной области памяти. При тестировании драйвера слежения мне иногда удавалось подвесить тестовое приложение во время его обращения к драйверу через IOCTL. Приложение больше не реагировало на запросы, и его невозможно было удалить из памяти. Что еще хуже, система не могла завершить работу. Такая ситуация раздражает почти так же, как BSOD!

Компоновка с системными модулями во время выполнения

После реализации основного интерфейса вызовов ядра следующая задача — отобразить символьные имена функций на линейные адреса, необходимые в ассемблерной инструкции CALL в листинге 6.2. Этот этап очень важен, потому что нельзя полагаться на то, что точки входа различных функций API ядра долго останутся неизменными. Где только возможно, функцию следует вызывать по имени. Обращаться к системной функции по адресу нужно только в особых случаях, как правило, только при вызове функций, которые не экспортируются в модуль, из которого осуществляется вызов. В большинстве случаев желательно применять символьное имя, расположенное в разделе экспорта модуля.

Определение имен, экспортируемых образом PE

Программист, работающий с Win32, постоянно сталкивается с задачей компоновки с экспортируемой DLL функцией на этапе выполнения. Например, если вы хотите написать библиотеку DLL, использующую расширенные возможности Windows 2000, но которая работала бы при этом с ограниченной функциональностью в устаревших системах, таких как Windows 95 или Windows 98, компоновку новых функций следует осуществлять во время выполнения, и если они недоступны, возвращаться к схеме работы по умолчанию. В этом случае нужно просто вызвать функцию GetModuleHandle(), если DLL уже находится в памяти и гарантированно останется в ней достаточно долго, или LoadLibrary(), если библиотеку нужно еще загрузить или защитить от преждевременной выгрузки из памяти. Полученный дескриптор модуля затем используется в нескольких последовательных вызовах функции GetProcAddress(), извлекающих точки входа всех требуемых приложению функций DLL. Кажется естественным применить этот подход и для функций ядра, экспортируемых ntoskrnl.exe, hal.dll или другими системными модулями. Однако ни одна из вышеупомянутых функций в этой ситуации работать не будет!

GetModuleHandle() сообщит, что модуль не загружен, а если передать GetProcAddress() жестко заданный дескриптор модуля, например такой, как (HMODULE) 0x80400000 для модуля ntoskrnl.exe, функция вернет NULL. Если подумать, такое поведение кажется разумным, ведь эти функции разработаны для компонентов Win32, работающих в режиме пользователя и загруженных в нижнюю половину 4 Гбайт линейного адресного пространства. Почему они должны знать о существовании компонентов режима ядра, которые в любом случае недостижимы для приложений Win32?

Если подсистема Win32 нам не поможет, нам следует предоставить эту работу драйверу режима ядра — стандартная стратегия, применяемая в данной книге. Недокументированная функция MmGetSystemRoutineAddress() из модуля ntoskrnl.exe решает задачу, но, к сожалению, она отсутствует в Windows NT 4.0. При создании программ-примеров для этой книги максимально возможная совместимость с предшественником Windows 2000 была одной из главных предпосылок, поэтому в этом случае я решил обойтись без помощи системы. В библиотеке времени выполнения Windows 2000 реализованы некоторые средства синтаксического разбора файла образа, такие как недокументированная функция RtlImageNtHeader(), чей прототип показан в листинге 6.4. Эта простая функция берет базовый адрес образа модуля, отображенного в линейную память (то есть указатель на ее структуру IMAGE_DOS_HEADER, что определено в заголовочном файле winnt.h комплекта Win32 SDK), и возвращает указатель на заголовок Portable PE, на который указывает член структуры заголовка DOS e_lfanew, расположенный в файле по смещению 0x3C. Эту функцию следует применять с осторожностью, так как в ней реализовано минимальное количество проверок корректности входного указателя. Проверяется только, не равен ли указатель NULL или 0xFFFFFFFF, а также содержит ли адресуемый им блок памяти в своем начале сигнатуру MZ. То есть если передать функции RtlImageNtHeader() неверный адрес, не равный ни NULL, ни 0xFFFFFFFF, BSOD появится сразу же, как только она начнет читать признак заголовка DOS. Странно, но Windows NT 4.0 выполняет этот код в оболочке SEH, а Windows 2000 —нет.

Листинг 6.4. Прототип функциии RtlImageNtHeader()

```
PIMAGE_NT_HEADERS NTAPI RtlImageHeader (PVOID Base);
```

Из листинга 6.4 видно, что RtlImageNtHeader() возвращает указатель на структуру IMAGE_NT_HEADERS. Полный набор структур данных файла PE определен в winnt.h. К сожалению, в заголовочных файлах комплекта DDK нет определений этих структур, поэтому необходимо добавить их самостоятельно. Определения структур данных, необходимых драйверу слежения для поиска идентификаторов (листинг 6.5), содержатся в его заголовочном файле w2k_spy.h. Структура IMAGE_NT_HEADERS образована просто из сигнатуры "PE\0\0" и структур IMAGE_FILE_HEADER и IMAGE_OPTIONAL_HEADER. Последняя завершается массивом структур IMAGE_DATA_DIRECTORY, нужным для быстрого поиска специальных разделов файла. Первый элемент массива, индекс которого обозначен как IMAGE_DIRECTORY_ENTRY в самом начале листинга 6.5, указывает на раздел экспорта с именами и адресами функций, экспортируемых модулем. Именно в этом разделе нужно искать имена функций, которые передаются интерфейсу вызовов ядра.

Листинг 6.5. Некоторые из основных структур файла PE

```
#define  IMAGE_DIRECTORY_ENTRY_EXPORT          0
#define  IMAGE_DIRECTORY_ENTRY_IMPORT          1
#define  IMAGE_DIRECTORY_ENTRY_RESOURCE        2
#define  IMAGE_DIRECTORY_ENTRY_EXCEPTION       3
#define  IMAGE_DIRECTORY_ENTRY_SECURITY        4
#define  IMAGE_DIRECTORY_ENTRY_BASERELOC       5
#define  IMAGE_DIRECTORY_ENTRY_DEBUG           6
#define  IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7
#define  IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8
#define  IMAGE_DIRECTORY_ENTRY_TLS             9
#define  IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10
#define  IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11
#define  IMAGE_DIRECTORY_ENTRY_IAT             12
#define  IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13
#define  IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14

#define  IMAGE_NUMBEROF_DIRECTORY_ENTRIES      16

//  ------------------------------------------------------

typedef struct _IMAGE_FILE_HEADER
      {
      WORD      Machine.
      WORD      NumberOfSections.
      DWORD     TimeDateStamp.
      DWORD     PointerToSymbolTable;
      DWORD     NumberOfSymbols;
      WORD      SizeOfOptionalHeader.
      WORD      Characteristics.
      }
      IMAGE_FILE_HEADER.  *PIMAGE_FILE_HEADER;

//  ------------------------------------------------------

typedef struct _IMAGE_DATA_DIRECTORY
      {
      DWORD     VirtualAddress;
      DWORD     Size.
      }
      IMAGE_DATA_DIRECTORY.  *PIMAGE_DATA_DIRECTORY;

//  ------------------------------------------------------

typedef struct _IMAGE_OPTIONAL_HEADER
      {
      WORD                  Magic.
      BYTE                  MajorLinkerVersion;
      BYTE                  MinorLinkerVersion;
      DWORD                 SizeOfCode.
      DWORD                 SizeOfInitializedData.
      DWORD                 SizeOfUninitializedData.
      DWORD                 AddressOfEntryPoint.
      DWDRD                 BaseOfCode;
      DWORD                 BaseOfData;
      DWORD                 ImageBase.
```

```
    DWORD                   SectionAlignment.
    DWORD                   FileAlignment.
    WORD                    MajorOperatingSystemVersion:
    WORD                    MinorOperatingSystemVersion:
    WORD                    MajorImageVersion.
    WORD                    MinorImageVersion.
    WORD                    MajorSubsystemVersion.
    WORD                    MinorSubsystemVersion:
    DWORD                   Win32VersionValue:
    DWORD                   SizeOfImage.
    DWORD                   SizeOfHeaders.
    DWORD                   CheckSum
    WORD                    Subsystem
    WORD                    DllCharacteristics:
    DWORD                   SizeOfStackReserve:
    DWORD                   SizeOfStackCommit:
    DWORD                   SizeOfHeapReserve.
    DWORD                   SizeOfHeapCommit:
    DWORD                   LoaderFlags
    DWORD                   NumberOfRvaAndSizes:
    IMAGE_DATA_DIRECTORY    DataDirectory
                            [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    }
    IMAGE_DPTIONAL_HEADER.  *PIMAGE_OPTIONAL_HEADER;

//  ------------------------------------------------------

typedef struct _IMAGE_NT_HEADERS
    {
    DWORD                   Signature:
    IMAGE_FILE_HEADER       FileHeader:
    IMAGE_OPTIONAL_HEADER   OptionalHeader.
    }
    IMAGE_NT_HEADERS.  *PIMAGE_NT_HEADERS:

//  ------------------------------------------------------

typedef struct _IMAGE_EXPORT_DIRECTORY
    {
    DWORD    Characteristics:
    DWORD    TimeDateStamp:
    WORD     MajorVersion.
    WDRD     MinorVersion.
    DWORD    Name
    DWORD    Base
    DWORD    NumberOfFunctions.
    DWORD    NumberOfNames.
    DWORD    AddressOfFunctions:
    DWORD    AddressOfNames:
    DWORD    AddressOfNameOrdinals:
    }
    IMAGE_EXPORT_DIRECTORY.  *PIMAGE_EXPORT_DIRECTORY.
```

Схема раздела экспорта в файле PE определяется структурой, расположенной в нижней части листинга 6.5. Главным образом она состоит из заголовка, составленного из членов IMAGE_EXPORT_DIRECTORY, трех массивов переменной длины и не-

скольких завершающихся нулем строк ANSI. Элемент экспорта, как правило, задается следующими тремя параметрами:

1. Символьное имя, завершающееся нулем и состоящее из 8-битных символов ANSI.
2. 16-битный порядковый номер.
3. 32-битное смещение по отношению к началу файла образа.

Механизм экспорта не ограничивается только функциями. Он просто является средством привязки идентификатора к адресу внутри образа PE. В случае функции идентификатор соответствует ее точке входа, в случае открытой переменной идентификатор ссылается на ее базовый адрес. Для назначения адреса характеристики идентификаторов заносятся в три независимых массива, обозначенных на рис. 6.1 как массив итоговых адресов, массив смещений имен и массив порядковых номеров. Эти массивы соответствуют членам структуры IMAGE_EXPORT_DIRECTORY AddressOfFunctions, AddressOfNames и AddressOfNameOrdinals, задающим смещения массивов по отношению к базовому адресу образа. В члене Name находится смещение символьной строки с именем самого файла PE. Если исполняемый файл будет переименован, эта строка поможет получить его исходное имя. Рисунок 6.1 представляет только общую схему расположения раздела экспорта, так как порядок массивов и подраздела символьных строк не фиксируется. При создании файла PE можно как угодно менять их местами, главное, чтобы члены структуры IMAGE_EXPORT_DIRECTORY ссылались на них правильно. То же самое относится и к строке, на которую ссылается член структуры Name. Хотя обычно строка с именем файла расположена в начале области строк имен, это необязательно. Никогда не стоит полагаться на то, что части раздела экспорта, расположение которых может меняться, находятся в каких-то конкретных местах.

Члены NumberOfFunctions и NumberOfNames структуры IMAGE_EXPORT_DIRECTORY определяют количество элементов соответственно в массивах AddressOfFunctions и AddressOfNames. Число элементов в массиве AddressOfNameOrdinals не отслеживается, поскольку оно всегда совпадает с количеством элементов массива AddressOfNames. Тот факт, что ведутся два отдельных счетчика элементов для адресов и имен, допускает возможность создания исполняемых файлов, экспортирующих неименованные адреса. Я такого файла никогда не видел, но при работе с массивами неплохо бы помнить об этой возможности. Опять же не стоит полагаться на допущения!

Процесс поиска адреса экспортируемой функции или переменной по их имени при заданном базовом адресе модуля (HMODULE на языке Win32) состоит из следующих этапов:

1. Вызвать RtlImageNtHeader(), передав ей базовый адрес модуля, и получить структуру модуля IMAGE_NT_HEADERS. Если функция вернет NULL, адресу не соответствует корректный образ PE.
2. Найти смещение раздела экспорта, используя константу IMAGE_DIRECTORY_ENTRY_EXPORT в качестве индекса массива DataDirectory, входящего в член OptionalHeaders структуры IMAGE_NT_HEADERS.
3. Определить расположение массива имен в разделе экспорта, прочитав значение члена AddressOfNames заголовка IMAGE_EXPORT_DIRECTORY.

4. Перебирать имена, пока не будет найдено нужное или не будет достигнут конец массива, заданный членом NumberOfNames.
5. Если имя найдено, прочитать его порядковый номер в массиве порядковых номеров по индексу, равному индексу в массиве имен. Порядковые номера начинаются с нуля, поэтому порядковый номер имени можно непосредственно использовать как индекс в массиве адресов.
6. Прибавить базовый адрес модуля к смещению, полученному из массива адресов.

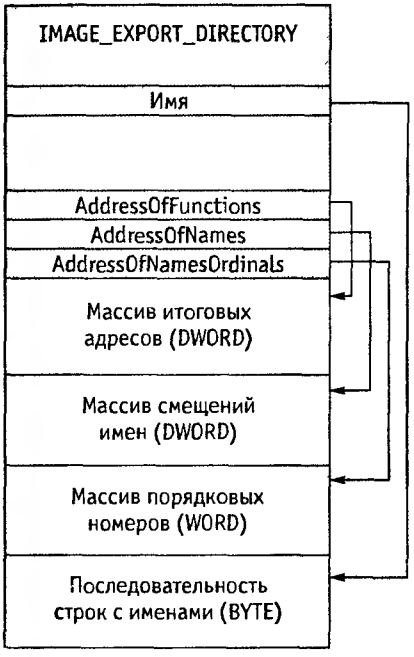

Рис. 6.1. Типичная схема раздела экспорта файла в формате PE

Последовательность шагов выглядит довольно простой. Однако в ней содержится одна неизвестная величина: базовый адрес модуля. В то время как вышеприведенные действия по существу отражают принцип работы функции Win32 GetProcAddress(), поиск адреса модуля должен имитировать действия функции GetModuleHandle(). При просмотре списка имен функций, экспортируемых ntoskrnl.exe, ни одно из них даже отдаленно не похоже на имя функции, способной выполнить эту задачу. Причина в том, что для этой и многих других задач, в которых требуется доступ к внутренним системным данным, в ядре Windows 2000 существует одна многоцелевая функция — NtQuerySystemInformation().

Поиск системных модулей и драйверов в памяти

NtQuerySystemInformation() — одна из ключевых функций API для системных программистов Windows 2000, вряд ли найдется какая-либо встроенная утилита администрирования, в которой бы она не использовалась, — однако в документации

Device DriverKit (DDK) о ней нет никаких упоминаний. Один раз она встречается в комментариях к структуре CONFIGURATION_INFORMATION в заголовочном файле ntddk.h, что подтверждает существование этой функции, но и все. Если бы существовал «коэффициент недокументируемости», определяемый как отношение полезности функции к частоте ее упоминания в документации Microsoft, NtQuerySystemInformation(), несомненно, заняла бы первое место. Помимо многих других замечательных способностей эта функция может получать список загруженных системных модулей, включая все ключевые компоненты системы и драйверы режима ядра.

В исходных файлах драйвера слежения содержится необходимый минимум кода и определений типов, требуемых для получения списка загруженных модулей через NtQuerySystemInformation(). С точки зрения вызывающей программы это достаточно простая функция. Она принимает четыре аргумента, как показано в листинге 6.6. Тип запрашиваемой информации определяется начинающейся с нуля числовой величиной SystemInformationClass. Информация, размер данных которой может варьироваться в зависимости от класса информации, копируется в буфер SystemInformation, переданный вызывающей программой. Длина буфера указана в аргументе SystemInformationLength. В случае успешного завершения фактическое число скопированных в буфер байт записывается в переменную, на которую указывает ReturnLength. Существует одна трудность: если буфер окажется недостаточно большим, функция не сообщает количество байт, которое ей нужно скопировать. Поэтому вызывающая программа должна путем метода проб и ошибок добиться того, чтобы код состояния изменился с STATUS_INFO_LENGTH_MISMATCH (0xC0000004) на STATUS_SUCCESS (0x00000000).

Листинг 6.6. Прототип функции NtQuerySystemInformation()

```
NTSTATUS NTAPI
ZwQuerySystemInformation (DWORD    SystemInformationClass,
                          PVOID    SystemInformation,
                          DWORD    SystemInformationLength,
                          DWORD    ReturnLength);
```

В листинге 6.6 показана не сама функция NtQuerySystemInformation(), а ее пара — ZwQuerySystemInformation(), которая отличается только префиксом. Как вам известно из главы 2, варианты функций Native API Nt* и Zw* работают одинаково при вызове из пользовательского режима. В реализующем интерфейс модуле ntdll.dll все пары функций проходят по одному и тому же маршруту — через INT 2Eh. В режиме ядра ситуация меняется. В этом случае вызовы Native API обрабатываются в ntoskrnl.exe и пути выполнения для функций Nt* и Zw* различаются. Варианты Zw* снова направляются через шлюз прерываний INT 2Eh, в точности как в ntdll.dll, а вот варианты Nt* обходят его. В глоссарии документации DDK Microsoft так описывает набор функций Zw*:

«Набор точек входа, соответствующий системным вызовам исполняющей системы. Обращение к точке входа ZwXxx из кода режима ядра (в том числе обращения из других системных вызовов или драйверов) соответствует обращению к системному вызову, за исключением того, что при этом не проверяется корректность прав доступа вызывающей программы и аргументов „псевдонима" Zw, и предыдущий режим не устанавливается в режим пользователя».

(Windows 2000 DDK\Kernel-Mode Drivers\Design Gide\Kernel-Mode Glossary\Z\Zw routines.)

Последнее упоминание о «предыдущем режиме» (previous mode) существенно. Питер Г. Вискарола (Peter G. Viscarola) и В. Энтони Мэйсон (W. Anthony Mason) объясняют это более понятно:

«Хотя из режима ядра обычно можно вызывать оба варианта функций, версия Zw применяется вместо Nt для того, чтобы установить предыдущий режим (а следовательно, и режим, в котором выполнялся запрос) в режим ядра». (Viscarola and Mason 1999, с.18)

Такая смена режимов обладает следующим побочным эффектом: вызов NtQuerySystemInformation() из драйвера режима ядра без дополнительных условий вернет код ошибки STATUS_ACCESS_VIOLATION (0xC0000005), а вызов ZwQuerySystemInformation() завершится успешно или по крайней мере вернет STATUS_LENGTH_MISMATCH.

В листинге 6.7 показаны определения констант и типов, требуемых для класса SystemModuleInformation. Список загруженных модулей возвращается в виде структуры MODULE_LIST, состоящей из 32-битного счетчика модулей и массива структур MODULE_INFO, по одной для каждого модуля.

Листинг 6.7. Определения SystemModuleInformation

```
#define SystemModuleInformation 11 // SYSTEMINFOCLASS

//  ------------------------------------------------------

typedef struct _MODULE_INFO
      {
      DWORD    dReserved1;
      DWORD    dReserved2;
      PVOID    pBase;
      DWORD    dSize;
      DWORD    dFlags;
      WORD     wIndex;
      WORD     wRank;
      WORD     wLoadCount;
      WORD     wNameOffset.
      BYTE abPath  [MAXIMUM_FILENAME_LENGTH];
      }
      MODULE_INFO. *PMODULE_INFO. **PPMODULE_INFO;

#define MODULE_INFO_ sizeof (MODULE_INFO)

//  ------------------------------------------------------

typedef struct _MODULE_LIST
      {
      DWORD         dModules;
      MDDULE_INFO   aModules [].
      }
      MDDULE_LIST. *PMODULE_LIST. **PPMODULE_LIST.

#define MODULE_LIST_ sizeof (MODULE_LIST)
```

Теперь все готово для вызова ZwQuerySystemInformation(). В листинге 6.8 приведена функция SpyModuleList(), выполняющая обычный цикл подбора нужного буфера для этой функции API, а также две простые функции управления памятью, SpyMemoryCreate() и SpyMemoryDestroy(), внутри которых осуществляется вызов функций из модуля Executive Windows 2000 ExAllocatePoolWithTag() и ExFreePool(). Подбор начинается с буфера объемом 4096 байт, и его размер все время удваивается, пока код возврата равен STATUS_INFO_LENGTH_MISMATCH. При любом другом коде возврата происходит выход из цикла. Необязательные аргументы pdData и pns содержат дополнительную информацию о возвращаемом значении. В случае неудачного завершения работы функции SpyModuleList() возвращает NULL, и тогда в буфере NTSTATUS, на который указывает pns, содержится код ошибки, а *pdData устанавливается в ноль. При успешном завершении *pdData показывает число скопированных в буфер байт, а *pns сообщает STATUS_SUCCESS.

Листинг 6.8. Получение списка модулей при помощи ZwQuerySystemInformation()

```
#define SPY_TAG '>YPS'    // SPY>, прочитанный сзади наперед

//  ------------------------------------------------------

PVOID SpyMemoryCreate (DWORD dSize)
      {
      return ExAllocatePoolWithTag (PagedPool, max (dSize, 1),
                                    SPY_TAG);
      }

//  ------------------------------------------------------

PVOID SpyMemoryDestroy (PVOID pData)
      {
      if (pData != NULL) ExFreePool (pData);
      return NULL;
      }

//  ------------------------------------------------------

PMODULE_LIST SpyModuleList (PDWORD     pdData,
                            PNTSTATUS  pns)
      {
      DWORD              dSize;
      DWORD              dData     = 0;
      NTSTATUS           ns        = STATUS_INVALID_PARAMETER;
      PMODULE_LIST       pml       = NULL;

      for (dSize = PAGE_SIZE;
           (pml == NULL) && dSize;
            dSize <<= 1)
          {
          if ((pml = SpyMemoryCreate (dSize)) == NULL)
              {
              ns = STATUS_NO_MEMORY;
              break;
              }
          ns = ZwQuerySystemInformation (SystemModuleInformation,
                                         pml, dSize, &dData);
```

```
        if (ns != STATUS_SUCCESS)
            {
            pml       = SpyMemoryDestroy (pml);
            dData     = 0;

            if (ns != STATUS_INFO_LENGTH_MISMATCH) break;
            }
        }
    if (pdData     != NULL) *pdData = dData;
    if (pns        != NULL) *pns  = ns;
    return pml;
    }
```

Для получения базового адреса заданного модуля осталось предпринять весьма простые действия. В листинге 6.9 определены две дополнительные функции: SpyModuleFind() является расширенной оболочкой SpyModuleList(), в ней осуществляется поиск заданного имени в возвращенном ZwQuerySystemInformation() списке модулей, а SpyModuleBase(), в свою очередь, образует оболочку над SpyModuleFind(), получая базовый адрес рассматриваемого модуля из его структуры MODULE_INFO и отбрасывая остальные поля структуры. Завершающая листинг 6.9 функция SpyModuleHeader() вызывает SpyModuleBase() и передает результат функции RtlImageNtHeader(), которая делает первый шаг к разделу экспорта загруженного модуля.

Листинг 6.9. Поиск информации об указанном модуле

```
PMODULE_LIST SpyModuleFind (PBYTE       pbModule,
                            PDWORD      pdIndex,
                            PNTSTATUS   pns)
    {
    DWORD               i;
    DWORD               dIndex    = -1;
    NTSTATUS            ns        = STATUS_INVALID_PARAMETER;
    PMODULE_LIST        pml       = NULL;

    if ((pml = SpyModuleList (NULL, &ns)) != NULL)
        {
        for (i = 0; i < pml->dModules; i++)
            {
            if (!_stricmp (pml->aModules [i].abPath +
                           pml->aModules [i].wNameOffset,
                           pbModule))
                {
                dIndex = i;
                break;
                }
            }
        if (dIndex == -1)
            {
            pml = SpyMemoryDestroy (pml);
            ns  = STATUS_NO_SUCH_FILE;
            }
        }
    if (pdIndex    != NULL) *pdIndex = dIndex;
    if (pns        != NULL) *pns   = ns;
    return pml;
    }
```

продолжение ⇩

Листинг 6.9 (*продолжение*)

```
//  ----------------------------------------------------------

PVOID SpyModuleBase (PBYTE          pbModule,
                     PNTSTATUS pns)
      {
      PMODULE_LIST   pml;
      DWORD          dIndex;
      NTSTATUS       ns        = STATUS_INVALID_PARAMETER.
      PVOID          pBase     = NULL;

      if ((pml = SpyModuleFind (pbModule, &dIndex, &ns))
           != NULL)
           {
           pBase = pml->aModules [dIndex].pBase;
           SpyMemoryDestroy (pml);
           }
      if (pns != NULL) *pns = ns;
      return pBase;
      }

//  ----------------------------------------------------------

PIMAGE_N  ADERS SpyModuleHeader (PBYTE      pbModule
                                 PPVOID     ppBase,
                                 PNTSTATUS  pns)
      {
      PVOID                  pBase     = NULL;
      NTSTATUS               ns        = STATUS_INVALID_PARAMETER;
      PIMAGE_NT_HEADERS      pinh      = NULL;

      if (((pBase = SpyModuleBase (pbModule, &ns))!= NULL)
           &&
           ((pinh  = RtlImageNtHeader (pBase))    == NULL))
           {
           ns = STATUS_INVALID_IMAGE_FORMAT;
           }
      if (ppBase  != NULL) *ppBase = pBase;
      if (pns     != NULL) *pns    = ns;
      return pinh;
      }
```

Определение адресов идентификаторов экспортируемых функций и переменных

В предыдущих разделах было объяснено, как осуществлять поиск символьного имени экспортируемой функции или переменной в файле образа формата PE и как определять базовый адрес загруженного системного модуля или драйвера. Давайте теперь соберем все части воедино. Поиск экспортируемого данным модулем идентификатора осуществляется в три этапа:

1. Найти линейный базовый адрес модуля.
2. Осуществить поиск идентификатора в разделе экспорта модуля.
3. Добавить смещение идентификатора к адресу модуля.

Первый шаг был весьма подробно обсужден выше. В листинге 6.10 приведены детали реализации остальных шагов. Функция SpyModuleExport() принимает в аргументе pbModule имя одного из системных файлов, такого как ntoskrnl.exe, hal.dll или ntfs.sys, и возвращает указатель на структуру модуля IMAGE_EXPORT_DIRECTORY при условии, что модуль находится в памяти ядра и у него есть раздел экспорта. Необязательные аргументы ppBase и pns возвращают дополнительную информацию: в случае успешного завершения работы функции *ppBase возвращает базовый адрес модуля, а pns* сообщает код ошибки при неудачном завершении. Сначала функция SpyModuleExport() вызывает SpyModuleHeader() для поиска структуры IMAGE_NT_HEADERS, после этого определяет расположение массива PE DataDirectory, в первом элементе которого содержатся характеристики раздела экспорта. Если член VirtualAddress этого элемента-структуры IMAGE_DATA_DIRECTORY (см. листинг 6.5) не равен NULL и член Size содержит приемлемое значение, то образ PE содержит раздел экспорта. В этом случае SpyModuleExport() добавляет базовый адрес модуля к значению VirtualAddress при помощи макроса PTR_ADD(), получая абсолютный линейный адрес структуры IMAGE_EXPORT_DIRECTORY. Иначе SpyModuleExport() возвращает NULL и устанавливает код возврата в STATUS_DATA_ERROR (0xC000003E).

Листинг 6.10. Поиск идентификаторов в разделе экспорта модуля

```
#define PTR_ADD(_base,_offset) \
              ((PVOID) ((PBYTE) (_base) + (DWORD) (_offset)))

//  ------------------------------------------------------

PIMAGE_EXPORT_DIRECTORY SpyModuleExport (PBYTE       pbModule,
                                         PPVOID      ppBase,
                                         PNTSTATUS   pns)
     {
     PIMAGE_NT_HEADERS         pinh;
     PIMAGE_DATA_DIRECTORY     pidd;
     PVOID                     pBase  = NULL;
     NTSTATUS                  ns     = STATUS_INVALID_PARAMETER;
     PIMAGE_EXPORT_DIRECTORY   pied   = NULL;

     if ((pinh = SpyModuleHeader (pbModule, &pBase, &ns))
          != NULL)
          {
          pidd = pinh->OptionalHeader.DataDirectory
                    + IMAGE_DIRECTORY_ENTRY_EXPORT;

          if (pidd->VirtualAddress &&
               (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
               {
               pied = PTR_ADD (pBase, pidd->VirtualAddress);
               }
          else
               {
               ns = STATUS_DATA_ERROR;
               }
          }
     if (ppBase != NULL) *ppBase = pBase;
```

продолжение

Листинг 6.10 (*продолжение*)

```
    if (pns   != NULL) *pns  = ns;
    return pied;
    }

//  ----------------------------------------------------------

PVOID SpyModuleSymbol (PBYTE       pbModule,
                       PBYTE       pbName,
                       PPVOID      ppBase,
                       PNTSTATUS   pns)
    {
    PIMAGE_EXPORT_DIRECTORY      pied,
    PDWORD                       pdNames, pdFunctions,
    PWORD                        pwOrdinals;
    DWORD                        i, j,
    PVOID                        pBase     = NULL;
    NTSTATUS                     ns        = STATUS_INVALID_PARAMETER,
    PVOID                        pAddress  = NULL,

    if ((pied = SpyModuleExport (pbModule, &pBase, &ns))
        != NULL)
        {
        pdNames       = PTR_ADD (pBase, pied->AddressOfNames);
        pdFunctions   = PTR_ADD (pBase, pied->AddressOfFunctions),
        pwOrdinals    = PTR_ADD (pBase, pied->AddressOfNameOrdinals),

        for (i = 0; i < pied->NumberOfNames; i++)
            {
            j = pwOrdinals [i];

            if (!strcmp (PTR_ADD (pBase, pdNames [i]),
                         pbName))
                {
                if (j < pied->NumberOfFunctions)
                    {
                    pAddress = PTR_ADD (pBase,
                                        pdFunctions [j]);
                    }
                break;
                }
            }
        if (pAddress == NULL)
            {
            ns = STATUS_PROCEDURE_NOT_FOUND;
            }
        }
    if (ppBase != NULL) *ppBase = pBase,
    if (pns   != NULL) *pns  = ns,
    return pAddress;
    }
```

SpyModuleSymbol() делает всю оставшуюся работу. Она обращается к различным элементам, показанным на рис. 6.1. После получения указателя на структуру IMAGE_EXPORT_DIRECTORY при помощи функции SpyModuleExport(), вычисляются линейные адреса массивов адресов, имен и порядковых номеров, снова при помо-

щи макроса PTR_ADD(). К счастью, формат файлов PE задает указатели на свои внутренние структуры данных все время в виде смещений от базового адреса образа, поэтому макрос PTR_ADD() — удобное универсальное средство, преобразующее такое смещение в линейный адрес. Важно отметить, какую роль при поиске адреса играет массив порядковых номеров. Если при поиске в массиве имен идентификатор был найден, переменная i будет содержать индекс (индексация с нуля) элемента массива, указывающего на этот идентификатор. Это значение нельзя непосредственно использовать для получения соответствующего адреса, необходимо осуществить преобразование при помощи массива порядковых номеров, что делается в строке кода j = pwOrdinals [i];. Полученный при этом порядковый номер j (нумерация тоже с нуля) и будет окончательным индексом, определяющим правильный адрес. Обратите внимание: порядковые номера — 16-битные величины, а другие два массива содержат 32-битные числа. Если переданный функции SpyModuleSymbol() через аргумент pbName идентификатор не будет найден, функция вернет указатель, равный NULL, и код возврата STATUS_PROCEDURE_NOT_FOUND (0xC000007A).

Хотя кажется, что SpyModuleSymbol() предоставляет все необходимое для вызова функций ядра по имени, я создал еще одну функцию-оболочку. В листинге 6.11 показано итоговое улучшение: функция SpyModuleSymbolEx() получает одну строку, составленную из имен модуля и идентификатора в виде "module!symbol", и находит соответствующие адреса при помощи SpyModuleSymbol(). Большая часть кода относится к разбору входной строки для отделения имени модуля от идентификатора. Если разделитель "!" отсутствует, SpyModuleSymbolEx() считает, что поиск следует вести в модуле ntoskrnl.exe, так как он, безусловно, вызывается чаще всего.

Листинг 6.11. Улучшенная функция поиска идентификаторов

```
PVOID SpyModuleSymbolEx (PBYTE      pbSymbol,
                         PPVOID     ppBase,
                         PNTSTATUS  pns)
    {
    DWORD          i;
    BYTE           abModule
                   [MAXIMUM_FILENAME_LENGTH] = "ntoskrnl.exe";
    PBYTE          pbName    = pbSymbol,
    PVOID          pBase     = NULL,
    NTSTATUS       ns        = STATUS_INVALID_PARAMETER;
    PVOID          pAddress  = NULL;

    for (i = 0;
          pbSymbol [i] && (pbSymbol [i] != '!'),
          i++);

    if (pbSymbol [i++])
        {
        if (i <= MAXIMUM_FILENAME_LENGTH)
            {
            strcpyn (abModule, pbSymbol, i),
            pbName = pbSymbol + i,
            }
```

продолжение

Листинг 6.11 (*продолжение*)

```
        else
            {
            pbName = NULL.
            }
        }
    if (pbName != NULL)
        {
        pAddress = SpyModuleSymbol (abModule. pbName. &pBase. &ns):
        }
    if (ppBase != NULL) *ppBase = pBase:
    if (pns    != NULL) *pns    = ns:
    return pAddress:
    }
```

Работа с пользовательским режимом

Теперь разработка интерфейса вызовов ядра практически завершена, по крайней мере в части, относящейся к режиму ядра. Давайте посмотрим, что у нас есть на данный момент:

- функция SpyCallEx() (листинг 6.3), принимающая управляющий блок SPY_CALL_INPUT, в котором задан нужный адрес и еще некоторые аргументы. Она вызывает функцию ядра по переданному адресу и возвращает все результаты в управляющем блоке SPY_CALL_OUTPUT;
- механизм поиска экспортируемых системных функций и переменных по имени, реализованный в функции SpyModuleSymbolEx() (листинг 6.11).

Осталось ответить на последний вопрос: как нам воспользоваться этими инструментами из приложений режима пользователя? Как всегда, ответ дает технология управления вводом-выводом устройства (IOCTL). В данный момент драйвер слежения предоставляет ряд функций IOCTL, реализованных в модуле w2k_spy_sys. В листинге 6.12 показана нужная нам часть функции SpyDispatcher() из листинга 4.7 главы 4.

Последнюю строку табл. 6.1 занимает функция SPY_IO_CALL, которая послужит нам мостиком к пользовательскому режиму. Остальные функции приведены здесь только для информации. Я подумал, что раз уж драйвер слежения имеет доступ к этим значимым данным, то было бы неплохо передать их также и приложению пользовательского режима. Как и в главах 4 и 5, все новые функции IOCTL будут кратко описаны далее.

Таблица 6.1. Функции IOCTL, связанные с интерфейсом вызовов ядра

Имя функции	ID	Код IOCTL	Описание
SPY_IO_MODULE_INFO	19	0x8000604C	Возвращает информацию о загруженных системных модулях
SPY_IO_PE_HEADER	20	0x80006050C	Возвращает данные структуры IMAGE_JNT_HEADERS
SPY_IO_PE_EXPORT	21	0x80006054C	Возвращает данные структуры IMAGE_EXPORT_DIRECTORY
SPY_IO_PE_SYMBOL	22	0x80006058C	Возвращает адрес экспортируемого системного идентификатора
SPY_IO_CALL	23	0x8000E05CC	Вызывает функцию из загруженного модуля

Листинг 6.12. Диспетчер команд драйвера слежения (частично)

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT   pDeviceContext,
                        DWORD             dCode,
                        PVOID             pInput,
                        DWORD             dInput,
                        PVOID             pOutput,
                        DWORD             dOutput,
                        PDWORD            pdInfo)
      {
      SPY_MEMORY_BLOCK        smb,
      SPY_PAGE_ENTRY          spe;
      SPY_CALL_INPUT          sci;
      PHYSICAL_ADDRESS        pa;
      DWORD                   dValue, dCount;
      BOOL                    fReset, fPause, fFilter, fLine;
      PVOID                   pAddress;
      PBYTE                   pbName,
      HANDLE                  hObject;
      NTSTATUS                ns = STATUS_INVALID_PARAMETER,

      MUTEX_WAIT  (pDeviceContext->kmDispatch);

      *pdInfo = 0;

      switch (dCode)
//  ============================================
// лишние функции IOCTL опущены (см. листинг 4.7)
//  ============================================
          case SPY_IO_MODULE_INFO:
               {
               if ((ns = SpyInputPointer (&pbName, pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputModuleInfo (pbName, pOutput, dOutput, pdInfo);
                    }
               break;
               }
          case SPY_IO_PE_HEADER:
               {
               if ((ns = SpyInputPointer (&pAddress, pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputPeHeader (pAddress, pOutput, dOutput, pdInfo)
                    }
               break;
               }
          case SPY_IO_PE_EXPORT:
               {
               if ((ns = SpyInputPointer (&pAddress, pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputPeExport (pAddress, pOutput, dOutput, pdInfo);
                    }
               break;
               }
          case SPY_IO_PE_SYMBOL:
```

продолжение

Листинг 6.12 (*продолжение*)

```
                {
                if ((ns = SpyInputPointer (&pbName, pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputPeSymbol (pbName, pOutput, dOutput, pdInfo);
                    }
                break;
                }
            case SPY_IO_CALL:
                {
                if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_, pInput, dInput))
                    == STATUS_SUCCESS)
                    {
                    ns = SpyOutputCall (&sci, pOutput, dOutput, pdInfo);
                    }
                break;
                }
//  ============================================
// лишние функции IOCTL опущены (см. листинг 4.7)
//  ============================================
            }
        MUTEX_RELEASE  (pDeviceContext->kmDispatch);
        return ns;
        }
```

Функция IOCTL SPY_IO_MODULE_INFO

Функция IOCTL SPY_IO_MODULE_INFO получает базовый адрес модуля и отправляет обратно структуру SPY_MODULE_INFO, если адрес указывает на корректный образ PE. Определения этой структуры и вспомогательной функции SpyOutputModuleInfo(), вызываемой функцией SpyDispatcher() в листинге 6.12, показаны в листинге 6.13. SpyOutputModuleInfo() основана на функции SpyModuleFind() (листинг 6.9), которая возвращает данные в виде структуры MODULE_INFO, полученные от ZwQuerySystemInformation(). MODULE_INFO преобразуется в формат SPY_MODULE_INFO и отправляется обратно вызывающей функции.

Листинг 6.13. Реализация SPY_IO_MODULE_INFO

```
typedef struct _SPY_MODULE_INFO
        {
        PVOID     pBase;
        DWORD     dSize;
        DWORD     dFlags;
        DWORD     dIndex;
        DWORD     dLoadCount;
        DWORD     dNameOffset;
        BYTE      abPath  [MAXIMUM_FILENAME_LENGTH];
        }
          SPY_MODULE_INFO,
         *PSPY_MODULE_INFO,
        **PPSPY_MODULE_INFO;

#define SPY_MODULE_INFO_ sizeof (SPY_MODULE_INFO)
```

```
//  -------------------------------------------------------

NTSTATUS SpyOutputModuleInfo (PBYTE    pbModule,
                              PVOID    pOutput,
                              DWORD    dOutput,
                              PDWORD   pdInfo)

    {
    SPY_MODULE_INFO     smi,
    PMODULE_LIST        pml,
    PMODULE_INFO        pmi,
    DWORD               dIndex;
    NTSTATUS            ns = STATUS_INVALID_PARAMETER,

    if ((pbModule != NULL)
         &&
         SpyMemoryTestAddress (pbModule)
         &&
        ((pml = SpyModuleFind (pbModule, &dIndex, &ns))
        != NULL))
        {
        pmi = pml->aModules + dIndex;

        smi pBase           = pmi->pBase,
        smi dSize           = pmi->dSize,
        smi dFlags          = pmi->dFlags;
        smi dIndex          = pmi->wIndex,
        smi dLoadCount      = pmi->wLoadCount;
        smi.dNameOffset     = pmi->wNameOffset;

        strcpyn (smi abPath, pmi->abPath,
                 MAXIMUM_FILENAME_LENGTH);

        ns = SpyOutputBinary (&smi, SPY_MODULE_INFO_,
                              pOutput, dOutput, pdInfo),

        SpyMemoryDestroy (pml),
        }
    return ns;
    }
```

Функция IOCTL SPY_IO_PE_HEADER

Функция IOCTL SPY_IO_PE_HEADER является просто оболочкой IOCTL для функции API ядра RtlImageNtHeader() из модуля ntoskrnl.exe, как видно из листинга 6.14. Так же как SPY_IO_MODULE_INFO, эта функция ожидает базовый адрес модуля. Возвращаемое значение представляет собой структуру модуля IMAGE_NT_HEADERS.

Листинг 6.14. Реализация SPY_IO_PE_HEADER

```
NTSTATUS SpyOutputPeHeader (PVOID    pBase,
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD   pdInfo)

    {
    PIMAGE_NT_HEADERS   pinh,
```

продолжение

Листинг 6.14 (*продолжение*)

```
    NTSTATUS            ns = STATUS_INVALID_PARAMETER.

    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
        ns = SpyOutputBinary (pinh, IMAGE_NT_HEADERS_,
                              pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

Функция IOCTL SPY_IO_PE_EXPORT

Функция IOCTL SPY_IO_PE_EXPORT интереснее предыдущей. Вкратце, она возвращает вызывающей программе структуру IMAGE_EXPORT_DIRECTORY, связанную с базовым адресом модуля. При внимательном взгляде на реализацию этой функции в листинге 6.15 становится заметно, что она сильно похожа на функцию SpyModuleExport() из листинга 6.10. Но SpyOutputPeExport(), помимо этого, выполняет и много другой работы. Причина этого в том, что в структуре IMAGE_EXPORT_DIRECTORY везде используется относительная адресация, как объяснялось ранее. После того как данные будут скопированы в отдельный буфер, значения всех смещений относительно базового адреса окажутся бесполезными для вызывающей программы, поскольку базовый адрес изменится. Без дополнительной информации из заголовка PE невозможно заново получить правильный базовый адрес. Избавляя вызывающую программу от этой избыточной работы, SpyOutputPeExport() преобразует все смещения, указывающие на структуры данных в разделе экспорта, в смещения относительно начала этого раздела, вычитая из них для этого значение члена VirtuaAddress структуры IMAGE_DATA_DIRECTORY, задающего смещение начала раздела экспорта от базового адреса образа. Элементы массива адресов необходимо обрабатывать по-другому, поскольку они ссылаются на другие разделы образа PE. Поэтому SpyOutputPeExport() преобразует их значения в абсолютные линейные адреса, добавляя базовый адрес образа.

Листинг 6.15. Реализация SPY_IO_PE_EXPORT

```
NTSTATUS SpyOutputPeExport (PVOID   pBase,
                            PVOID   pOutput,
                            DWORD   dOutput,
                            PDWORD  pdInfo)
    {
    PIMAGE_NT_HEADERS          pinh;
    PIMAGE_DATA_DIRECTORY      pidd;
    PIMAGE_EXPORT_DIRECTORY    pied;
    PVOID                      pData;
    DWORD                      dData, dBias, i;
    PDWORD                     pdData;
    NTSTATUS                   ns = STATUS_INVALID_PARAMETER;

    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
```

```
        pidd = pinh->OptionalHeader.DataDirectory
               + IMAGE_DIRECTORY_ENTRY_EXPORT;

        if (pidd->VirtualAddress &&
            (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
            {
            pData = (PBYTE) pBase + pidd->VirtualAddress;
            dData = pidd->Size;

            if ((ns = SpyOutputBinary (pData, dData,
                                       pOutput, dOutput, pdInfo))
                == STATUS_SUCCESS)
                {
                pied  = pOutput;
                dBias = pidd->VirtualAddress;

                pied->Name                  -= dBias;
                pied->AddressOfFunctions    -= dBias;
                pied->AddressOfNames        -= dBias;
                pied->AddressOfNameOrdinals -= dBias;

                pdData = PTR_ADD (pied, pied->AddressOfFunctions);

                for (i = 0;
                     i < pied->NumberOfFunctions;
                     i++)
                    {
                    pdData [i] += (DWORD) pBase;
                    }
                pdData = PTR_ADD (pied, pied->AddressOfNames);

                for (i = 0; i < pied->NumberOfNames; i++)
                    {
                    pdData [i] -= dBias;
                    }
                }
            }
        else
            {
            ns = STATUS_DATA_ERROR;
            }
        }
    return ns;
    }
```

Функция IOCTL SPY_IO_PE_SYMBOL

Функция IOCTL SPY_IO_PE_SYMBOL обеспечивает доступ приложениям пользовательского режима к поисковому механизму интерфейса вызовов API. Ее реализация в листинге 6.18 не особенно впечатляет, так как она всего лишь оболочка IOCTL для функции SpyModuleSymbolEx() из листинга 6.11. Вызывающая программа должна передать указатель на строку в форме "module!symbol" или просто "symbol", если идентификатор следует искать в разделе экспорта модуля ntoskrnl.exe, и получить обратно указатель на линейный адрес, сопоставленный идентификатору, или NULL, если идентификатор некорректен или произошла ошибка.

Листинг 6.16. Реализация SPY_IO_PE_SYMBOL

```
NTSTATUS SpyOutputPeSymbol (PBYTE    pbSymbol
                            PVOID    pOutput,
                            DWORD    dOutput,
                            PDWORD   pdInfo)
    {
    PVOID     pAddress;
    NTSTATUS ns = STATUS_INVALID_PARAMETER;

    if ((pbSymbol != NULL)
        &&
        SpyMemoryTestAddress  (pbSymbol)
        &&
        ((pAddress = SpyModuleSymbolEx (pbSymbol, NULL, &ns))
         != NULL))
        {
        ns = SpyOutputPointer (pAddress,
                               pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

Функция IOCTL SPY_IO_CALL

И наконец, долгожданная функция IOCTL SPY_IO_CALL. В листинге 6.17 показаны детали реализации. Если переданный ей адрес строки идентификатора корректен, функция вызывает SpyModuleSymbolEx() и затем, если идентификатору был сопоставлен адрес, вызывает SpyCallEx(). Так же как и SPY_IO_PE_SYMBOL, эта функция ожидает, что имя идентификатора будет задано в виде "module!symbol" или просто "symbol" для модуля ntoskrnl.exe. На этот раз только определяющая идентификатор строка передается в функцию как часть должным образом инициализированной структуры SPY_CALL_INPUT. В случае успешного завершения функция SPY_IO_CALL возвращает структуру SPY_CALL_OUTPUT, содержащую либо результат вызова функции, если идентификатору соответствует функция API, либо значение переменной, если идентификатор задает открытую переменную, такую как NtBuildNumber или KeServiceDescriptorTable.

В случае неудачного завершения SPY_IO_CAL не возвращает никаких данных и вызывающая программа должна правильно обрабатывать эту ситуацию. Если никак не прореагировать на эту ошибку, это будет означать получение фиктивных данных от функции ядра. Если эти данные затем будут переданы другой функции ядра, могут возникнуть проблемы. Если вам повезет, ошибка данных будет перехвачена обработчиком исключений внутри SpyCallEx(), если же нет, то при обращении к IOCTL драйвера слежения может зависнуть весь процесс. Как обычно, может появиться BSOD. Но не беспокойтесь — в следующем разделе я покажу, как правильно работать с интерфейсом вызовов ядра из приложения пользовательского режима.

Листинг 6.17. Реализация SPY_IO_CALL

```
NTSTATUS SpyOutputCall (PSPY_CALL_INPUT   psci,
                        PVOID             pOutput,
                        DWORD             dOutput,
                        PDWORD            pdInfo)
```

```
{
SPY_CALL_OUTPUT     sco;
NTSTATUS            ns = STATUS_INVALID_PARAMETER;

if (psci->pbSymbol != NULL)
     {
     psci->pEntryPoint =
          (SpyMemoryTestAddress  (psci->pbSymbol)
          ? SpyModuleSymbolEx    (psci->pbSymbol, NULL, &ns)
          : NULL);
     }
if ((psci->pEntryPoint != NULL)                 &&
     SpyMemoryTestAddress  (psci->pEntryPoint)   &&
     ((ns = SpyCallEx (psci, &sco)) == STATUS_SUCCESS))
     {
     ns = SpyOutputBinary (&sco, SPY_CALL_OUTPUT_,
                           pOutput, dOutput, pdInfo);
     }
return ns;
}
```

Помещение интерфейса вызовов в DLL

Хотя теперь w2k_spy.sys экспортирует интерфейс вызовов для функций ядра, работать с этим интерфейсом не очень удобно. Предположим, требуется вызвать какую-нибудь простую функцию, такую как MmGetPhysicalAddress() или MmIsAddressValid(). Сначала необходимо заполнить структуру SPY_CALL_INPUT информацией о функции и ее аргументах. Затем вызвать функцию Win32 DeviceIoControl(). Если она вернет ERROR_SUCCESS, нужно получить данные из структуры SPY_CALL_OUTPUT, в ином случае — правильно обработать ошибку. Не очень привлекательная перспектива, не так ли? К счастью, в нашем распоряжении имеются библиотеки DLL, и если спрятать механизм IOCTL в DLL, делающую всю грязную работу, это решит проблему. Для этого предназначена библиотека w2k_call.dll на прилагающемся к книге компакт-диске. Приведенные в этом разделе фрагменты кода взяты из файлов w2k_call.c и w2k_call.h, расположенных в каталоге \src\w2k_call компакт-диска.

Обработка вызовов функций IOCTL

Прежде всего необходимо заключить вызовы DeviceIoControl() в удобную оболочку, поскольку через эту функцию обязаны проходить все обращения к функциям ядра. В листинге 6.18 показана такая функция-оболочка w2kSpyControl(), выполняющая следующие задачи:

- проверяет входные и выходные параметры;
- загружает драйвер слежения и открывает его, если это еще не сделано;
- вызывает DeviceIoControl();
- проверяет, соответствует ли размер выходных данных ожидаемому;
- устанавливает соответствующим образом код последней ошибки Win32.

В случае успешного завершения код последней ошибки системы, получаемой приложением при помощи функции GetLastError(), устанавливается в ERROR_SUCCESS(0). Иначе он устанавливается по следующим правилам:

- если некорректны входные или выходные параметры, код последней ошибки равен ERROR_INVALID_PARAMETER(87), то есть «The parameter is incorrect» (параметр некорректен), как следует из заголовочного файла winerror.h комплекта Platform Software Developmrnt Kit (SDK);
- если драйвер слежения не смог запуститься, кодом последней ошибки будет ERROR_GEN_FAILURE(31), то есть «A device attached to the system is not functioning» (присоединенное к системе устройство не работает);
- если размер данных, возвращенных драйвером слежения, не соответствует размеру буфера вызывающей программы, кодом последней ошибки будет ERROR_DATATYPE_MISMATCH, что означает «Data supplied is of wrong type» (неверен тип переданных данных);
- во всех остальных случаях w2kSpyControl() сохраняет значение последней ошибки, установленное функцией DeviceIoControl(), каким бы оно ни было. Как правило, это код NTSTATUS, возвращенный драйвером слежения, но преобразованный в более-менее соответствующий код состояния Win32.

Листинг 6.18. Основная оболочка функции DeviceIoControl()

```
BOOL WINAPI w2kSpyControl (DWORD  dCode,
                           PVOID  pInput,
                           DWORD  dInput,
                           PVOID  pOutput,
                           DWORD  dOutput)
     {
     DWORD    dInfo     = 0;
     BOOL fOk           = FALSE;

     SetLastError  (ERROR_INVALID_PARAMETER);

     if (((pInput  != NULL) || (!dInput )) &&
         ((pOutput != NULL) || (!dOutput)))
         {
         if (w2kSpyStartup (FALSE, NULL))
             {
             if (DeviceIoControl (ghDevice   dCode,
                                  pInput,    dInput,
                                  pOutput,   dOutput,
                                  &dInfo,    NULL))
                 {
                 if (dInfo == dOutput)
                     {
                     SetLastError  (ERROR_SUCCESS);
                     fOk = TRUE;
                     }
                 else
                     {
                     SetLastError  (ERROR_DATATYPE_MISMATCH);
                     }
                 }
             }
         else
```

```
            {
            SetLastError  (ERROR_GEN_FAILURE);
            }
        }
    return fOk.
    }
```

В листинге 6.18 заслуживает внимания вызов w2kSpyStartup(), осуществляемый непосредственно перед вызовом DeviceIoControl(). Поскольку библиотека w2k_call.dll основана на функциях драйвера режима ядра, этот драйвер должен быть каким-то образом помещен в память перед первым обращением к IOCTL. Более того, дескриптор устройства должен быть открыт, что даст возможность обратиться к устройству через DeviceIoControl(). Чтобы сохранить гибкость DLL, я выбрал смешанную модель, в которой вызывающая программа может либо полностью самостоятельно управлять загрузкой/выгрузкой и открытием/закрытием устройства, либо полагаться на механизм по умолчанию, предоставляя DLL самой управлять устройством. Автоматический режим работает весьма просто: загрузка драйвера и открытие устройства откладывается до первого обращения к IOCTL. При выгрузке DLL автоматически закрывает дескриптор устройства, однако сохраняет в памяти драйвер режима ядра. Последнее решение реализует безопасную стратегию. Поскольку вызывающая программа не предоставляет никакой информации о способе работы с драйвером, w2k_call.dll предполагает, что с ним могут работать и другие клиенты, поэтому выгрузка драйвера нарушила бы работу других приложений. Как объяснялось в главе 4 при обсуждении приложения просмотра памяти, проблема не в процессах, у которых до сих пор есть открытые дескрипторы устройства слежения. Диспетчер управления системными вызовами Windows 2000 отложит завершение работы драйвера до тех пор, пока все дескрипторы не будут закрыты. Проблема в том, что диспетчер запретит открытие каких-либо новых дескрипторов устройства.

Клиентское приложение библиотеки w2k_call.dll может управлять состоянием устройства слежения при помощи пары функций API w2kSpyStartup() и w2kSpyCleanup(), показанных в листинге 6.19. Поскольку эти функции могут вызываться параллельно в многопоточной среде, для монопольного доступа они используют критическую секцию. Только один поток единовременно может загружать/открывать или закрывать/выгружать драйвер слежения. Если, например, два потока вызовут w2kSpyStartup() приблизительно в одно и то же время, только одному из них будет разрешено открыть устройство. Другой поток будет приостановлен, и после возобновления ему будет предоставлено готовое к работе устройство.

Листинг 6.19. Функции управления устройством слежения

```
BOOL WINAPI w2kSpyLock (VOID)
    {
    BOOL fOk = FALSE;

    if (gpcs != NULL)
        {
        EnterCriticalSection (gpcs);
        fOk = TRUE;
        }
    return fOk;
    }
```

продолжение

Листинг 6.19 (*продолжение*)

```
//  ------------------------------------------------------

BOOL WINAPI w2kSpyUnlock (VOID)
      {
      BOOL fOk = FALSE;

      if (gpcs != NULL)
           {
           LeaveCriticalSection (gpcs);
           fOk = TRUE;
           }
      return fOk;
      }

//  ------------------------------------------------------

BOOL WINAPI w2kSpyStartup (BOOL        fUnload,
                           HINSTANCE  hInstance)
      {
      HINSTANCE hInstance1;
      SC_HANDLE hControl;
      BOOL      fOk = FALSE;

      w2kSpyLock ();

      hInstance1 = (hInstance != NULL
                         ? hInstance
                         : ghInstance);

      if ((ghDevice == INVALID_HANDLE_VALUE) &&
           w2kFilePath (hInstance1, awSpyFile,
                        awDriver, MAX_PATH)
           &&
           ((hControl = w2kServiceLoad (awSpyDevice,
                                        awSpyDisplay,
                                        awDriver, TRUE))
            != NULL))
           {
           ghDevice = CreateFile (awSpyPath,
                           GENERIC_READ     | GENERIC_WRITE,
                           FILE_SHARE_READ | FILE_SHARE_WRITE,
                           NULL, OPEN_EXISTING,
                           FILE_ATTRIBUTE_NORMAL, NULL);

           if ((ghDevice == INVALID_HANDLE_VALUE) && fUnload)
                {
                w2kServiceUnload (awSpyDevice, hControl);
                }
           else
                {
                w2kServiceDisconnect (hControl);
                }
           }
      fOk = (ghDevice != INVALID_HANDLE_VALUE);

      w2kSpyUnlock ();
      return fOk;
      }
```

```
//  ---------------------------------------------------

BOOL WINAPI w2kSpyCleanup (BOOL fUnload)
      {
      BOOL fOk = FALSE;

      w2kSpyLock ();

      if (ghDevice != INVALID_HANDLE_VALUE)
           {
           CloseHandle (ghDevice);
           ghDevice = INVALID_HANDLE_VALUE;
           }
      if (fUnload)
           {
           w2kServiceUnload (awSpyDevice, NULL);
           }
      w2kSpyUnlock ();
      return fOk;
      }
```

Специальные функции интерфейса вызовов

Вызовы DeviceIoControl() и автоматическое управление устройством слежения теперь осуществляются через набор функций с общей точкой входа в w2kSpyControl(). Следующий шаг — реализовать функции для обращения к SPY_IO_CALL устройства. В листинге 6.20 в основном показана реализация части пользовательского режима интерфейса вызовов ядра, представленная функциями w2kCallExecute(), w2kCall() и w2kCallV(). Судя по входным параметрам, первая функция представляет собой эквивалент режима пользователя функции SpyCallEx() из листинга 6.3. Реализация w2kCallExecute() показывает, что она обращается к функции драйвера слежения SPY_IO_CALL через w2kSpyControl(), проверив сначала, что входной управляющий блок содержит либо строку с идентификатором, либо адрес точки входа. Из листинга 6.12 известно, что SPI_IO_CALL реализована функцией SpyOutputCall() (листинг 6.17), которая, в свою очередь, основывается на SpyModuleSymbolEx() и SpyCallEx().

Листинг 6.20. Основные функции интерфейса вызовов

```
BOOL WINAPI w2kCallExecute (PSPY_CALL_INPUT   psci,
                            PSPY_CALL_OUTPUT  psco)
      {
      BOOL fOk = FALSE;

      SetLastError  (ERROR_INVALID_PARAMETER);

      if (psco != NULL)
           {
           psco->ullResult.QuadPart = 0;

           if ((psci != NULL)
                &&
                ((psci->pbSymbol     != NULL) ||
                 (psci->pEntryPoint != NULL)))
                {
```

продолжение ↓

Листинг 6.20 (*продолжение*)

```
            fOk = w2kSpyControl (SPY_IO_CALL,
                                 psci, SPY_CALL_INPUT_,
                                 psco, SPY_CALL_OUTPUT_);
            }
        }
    return fOk;
    }

//  ------  ----------------------------------------------

BOOL WINAPI w2kCall (PULARGE_INTEGER    puliResult
                     PBYTE    pbSymbol,
                     PVOID    pEntryPoint,
                     BOOL     fFastCall,
                     DWORD    dArgumentBytes,
                     PVOID    pArguments)
    {
    SPY_CALL_INPUT      sci;
    SPY_CALL_OUTPUT     sco;
    BOOL                fOk  = FALSE;

    sci fFastCall           = fFastCall,
    sci dArgumentBytes      = dArgumentBytes,
    sci.pArguments          = pArguments;
    sci.pbSymbol            = pbSymbol,
    sci.pEntryPoint         = pEntryPoint;

    fOk = w2kCallExecute (&sci, &sco);

    if (puliResult != NULL) *puliResult = sco uliResult,
    return fOk,
    }

//  ----------------------------------------------------------

BOOL WINAPI w2kCallV (PULARGE_INTEGER  puliResult
                      PBYTE            pbSymbol,
                      BOOL             fFastCall,
                      DWORD            dArgumentBytes,
                       . )
    {
    return w2kCall (puliResult, pbSymbol  NULL, fFastCall,
                          dArgumentBytes, &dArgumentBytes + 1),
    }
```

Функции SpyCall() и w2kCallV() из листинга 6.20 — это ключевые функции интерфейса вызовов ядра в библиотеке w2k_call.dll, образующие основу для нескольких более специализированных функций. Основная задача w2kCall() — заполнить структуру SPY_INPUT_CALL значениями переданных ей аргументов перед вызовом w2kCallExecute() и в качестве возвращаемого значения передать полученную величину типа ULARGE_INTEGER. Как объяснялось раньше, результат не обязательно должен содержаться во всех битах этой структуры, это зависит от типа возвращаемого значения вызванной функции ядра. Функция w2kCallV() является простой

оболочкой w2kCall(), обладая переменным списком параметров (поэтому ее имя завершается буквой V, variable). Поскольку список параметров функции w2kCall() настроен на самый общий случай вызовов функций API ядра, для многих распространенных видов функций он избыточен. Самый распространенный вид функции — это функция с соглашением о вызовах __stdcall (или NTAPI), возвращающая значение NTSTATUS. В этом случае аргумент fFastCall всегда равен FALSE и возвращаемые данные занимают только младшую половину 64-битной величины типа ULARGE_INTEGER. Поэтому функция w2kCallNT() из листинга 6.21 справляется с работой гораздо лучше. Обратите внимание на то, как w2kCallNT() обрабатывает ошибки, возвращаемые w2kCall(). Если w2kCall() возвращает FALSE, это значит, что w2kSpyControl() завершилась неудачно и результат вызова функции некорректен. В этом случае нет смысла получать значение младшей половины (LowPart) структуры uliResult, поскольку в нем содержится непредсказуемый мусор. Поэтому w2kCallNT() возвращает значение STATUS_IO_DEVICE_ERROR (0xC0000185). В конце концов, вызывающая функция должна быть готова к возврату значений, отличающихся от STATUS_SUCCESS (0x00000000), поэтому возврат этого кода ошибки выглядит приемлемым решением. Для других функций ядра, не сообщающих код NTSTATUS, необходим гораздо более осторожный отбор возвращаемых по умолчанию значений в случае неудачного завершения.

Листинг 6.21. Упрощенный интерфейс для видов функций NTAPI/NTSTATUS

```
NTSTATUS WINAPI w2kCallNT (PBYTE pbSymbol,
                           DWORD dArgumentBytes,
                           )

    {
    ULARGE_INTEGER uliResult.

    return (w2kCall (&uliResult, pbSymbol, NULL, FALSE,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR).
    }
```

В листинге 6.22 собраны пять дополнительных функций интерфейса для функций API с соглашением __stdcall, возвращающих основные типы данных BYTE, WORD, DWORD, DWORDLONG и PVOID. Число, которым заканчивается имя функции, показывает количество значащих бит в возвращаемом значении. w2kCallP() эквивалентно w2kCall32(), за исключением того, что 32-битное возвращаемое значение преобразуется в указатель операцией приведения типа. Нет необходимости реализовывать отдельные функции для работы с версиями основных типов данных со знаком или указателями различных типов, поскольку эти незначительные различия оставлены автоматическому приведению типов, осуществляемому компилятором. Заметьте, что все функции в листинге 6.22 ожидают, что возвращаемое по умолчанию значение будет передано им в качестве первого параметра. Это необходимо, поскольку интерфейс вызовов не знает, какое значение лучше будет вернуть в случае неудачного завершения функции режима ядра, поэтому за это отвечает вызывающая программа.

Листинг 6.22. Дополнительные функции интерфейса для распространенных типов функций

```
BYTE WINAPI w2kCall08 (BYTE   bDefault,
                       PBYTE  pbSymbol,
                       BOOL   fFastCall,
                       DWORD  dArgumentBytes,
                       ...)
      {
      ULARGE_INTEGER uliResult.

      return (w2kCall (&uliResult, pbSymbol,
                       NULL, fFastCall,
                       dArgumentBytes, &dArgumentBytes + 1)

               ? (BYTE) uliResult.LowPart
               : bDefault);
      }

//  ------------------------------------------------------

WORD WINAPI w2kCall16 (WORD   wDefault,
                       PBYTE  pbSymbol,
                       BOOL   fFastCall,
                       DWORD  dArgumentBytes,
                              ...)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCall (&uliResult, pbSymbol, NULL,
                       fFastCall, dArgumentBytes,
                       &dArgumentBytes + 1)

               ? (WORD) uliResult.LowPart
               : wDefault);
      }

//  ------------------------------------------------------

DWORD WINAPI w2kCall32 (DWORD  dDefault,
                        PBYTE  pbSymbol,
                        BOOL   fFastCall,
                        DWORD  dArgumentBytes,
                        ...)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCall (&uliResult, pbSymbol, NULL,
                fFastCall, dArgumentBytes, &dArgumentBytes + 1)

               ? uliResult.LowPart
               : dDefault);
      }

//  ------------------------------------------------------

QWORD WINAPI w2kCall64 (QWORD  qDefault,
                        PBYTE  pbSymbol,
                        BOOL   fFastCall,
```

```
                    DWORD  dArgumentBytes,
                    ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL,
                    fFastCall,
            dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.QuadPart
            : qDefault);
    }

//  ------------------------------------------------------

PVOID WINAPI w2kCallP (PVOID  pDefault,
                    PBYTE  pbSymbol,
                    BOOL   fFastCall,
                    DWORD  dArgumentBytes,
                    ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL,
                    fFastCall,
                    dArgumentBytes, &dArgumentBytes + 1)

            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

Функции интерфейса для копирования данных

Перед тем как перейти к более интересной задаче определения замен для нескольких настоящих функций ядра, нам потребуются несколько дополнительных строк стереотипного кода. Ранее я уже говорил, что интерфейс вызовов ядра устройства слежения может также работать с открытыми переменными, экспортируемыми модулями ядра. В описании листинга 6.2, в котором была показана функция SpyCall(), я объяснил, что отрицательное значение размера стека аргументов, передаваемое через член dArgumentBytes структуры SPY_CALL_INPUT, интерпретируется как дополнение до единицы размера экспортируемой переменной. В нашем случае SpyCall() не вызывает функцию по указанной точке входа, а копирует соответствующее число байт, начиная с этого адреса, в буфер результата. Если член dArgumentBytes равен –1, дополнение до единицы этого значения равно нулю и в буфер копируется сам адрес точки входа.

В листинге 6.23 показаны функции копирования данных, экспортируемые библиотекой w2k_call.dll. Просматривается явное соответствие этого набора функций набору функций интерфейса вызовов из листинга 6.22. Однако функциям из листинга 6.23 требуется меньше входных параметров. Для копирования значения экспортируемой переменной требуется одно лишь имя переменной, не нужны никакие входные параметры и соглашения о вызовах.

Листинг 6.23. Функции интерфейса для кодирования основных типов данных

```
BOOL WINAPI w2kCopy (PULARGE_INTEGER  puliResult,
                     PBYTE            pbSymbol,
                     PVOID            pEntryPoint,
                     DWORD            dBytes)
      {
      return w2kCall (puliResult, pbSymbol, pEntryPoint,
                      FALSE, 0xFFFFFFFF - dBytes, NULL);
      }

//  ------------------------------------------------------

BYTE WINAPI w2kCopy0B (BYTE   bDefault,
                       PBYTE  pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 1)
               ? (BYTE) uliResult.LowPart
               : bDefault);
      }

//  ------------------------------------------------------

WORD WINAPI w2kCopy16 (WORD   wDefault,
                       PBYTE  pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 2)
               ? (WDRD) uliResult.LowPart
               : wDefault);
      }

//  ------------------------------------------------------

DWORD WINAPI w2kCopy32 (DWORD dDefault,
                        PBYTE pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
               ? uliResult.LowPart
               : dDefault);
      }

//  ------------------------------------------------------

QWORD WINAPI w2kCopy64 (QWORD qDefault,
                        PBYTE pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 8)
               ? uliResult.QuadPart
               : qDefault);
      }
```

```
//  ------------------------------------------------------

PVOID WINAPI w2kCopyP (PVOID  pDefault,
                       PBYTE  pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
               ? (PVOID) uliResult.LowPart
               : pDefault);
      }

//  ------------------------------------------------------

PVOID WINAPI w2kCopyEP (PVOID pDefault,
                        PBYTE pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kCopy (&uliResult, pbSymbol, NULL, 0)
               ? (PVOID) uliResult.LowPart
               : pDefault);
      }
```

В листинге 6.23 основную часть работы выполняет w2kCopy(), во многом похожая на функцию w2kCall(). Здесь опять w2k_call.dll предоставляет отдельные функции для основных типов данных BYTE, WORD, DWORD, DWORDLONG и PVOID, заканчивающиеся числом значащих бит в возвращаемом значении. w2kCopyP() возвращает указатель, а w2kcopyEP() обрабатывает особый случай запроса адреса точки входа. Вызов w2kCopyEP() эквивалентен вызову функции драйвера слежения SPY_IO_PE_SYMBOL. Это, конечно, избыточно, но два различных пути к цели всегда лучше, чем вообще ни одного, не так ли?

Реализация функций-посредников API ядра

На данный момент у нас есть основной каркас для простой и легкой реализации замен функций API ядра. Я называю такие заменяющие функции термином «thunks» (посредники), как обычно[1] на языке Windows называется короткий фрагмент кода, служащий клиентом функции, реализованной в другой части системы. Также распространен термин «proxy» (заместитель), но он вызывает слишком стойкие ассоциации с компонентной моделью объектов (Component Object Model, COM) Microsoft, что могло бы сбивать с толку. Давайте начнем с двух очень простых функций диспетчера памяти Windows 2000, которые я использовал в качестве первых объектов для тестирования при разработке модуля w2k_call.dll: MmGetPhysicalAddress() и MmIsAddressValid(). В листинге 6.24 показано, как их функции-посредники реализованы при помощи w2kCall64() и w2kCall08(). Чтобы избежать путаницы с исходными функциями, ко всем функциям-посредникам я добавил символ подчеркивания.

[1] «Обычно» относится к английскому термину «thunk», в русской терминологии нет однозначного эквивалента. Вариант «шлюз» был бы не очень хорош в данной книге, поскольку перекликается с «gate». — *Примеч. перев.*

Листинг 6.24. Образец функций-посредников для MmGetPhysicalAddress() и MmIsAddressValid()

```
PHYSICAL_ADDRESS WINAPI
_MmGetPhysicalAddress (PVOID BaseAddress)
      {
      PHYSICAL_ADDRESS pa.

      pa.QuadPart = w2kCall64 (0, "MmGetPhysicalAddress",
                               FALSE, 4, BaseAddress);
      return pa;
      }
//  ----------------------------------------------------
BOOLEAN WINAPI
_MmIsAddressValid (PVOID VirtualAddress)
      {
      return w2kCall08 (FALSE, "MmIsAddressValid", FALSE,
                        4, VirtualAddress);
      }
```

MmGetPhysicalAddress() принимает 32-битный линейный адрес и возвращает 64-битную структуру данных PHISICAL_ADDRESS, которая представляет собой не что иное, как LARGE_INTEGER. Поэтому код функции-посредника вызывает w2kCall64(), что означает передачу 4 байт через стек аргументов и помещение параметра BaseAddress в список аргументов. В случае критичной ошибки IOCTL возвращается ноль (значение по умолчанию), что возвращает в случае ошибки и исходная функция. Параметр fFastCall равен FALSE, так как MmGetPhysicalAddress() работает по соглашению __stdcall. Функция-посредник MmIsAddressValid() реализована аналогично, за тем исключением, что возвращаются только восемь младших бит результата SpyCallEx(), содержащих величину с типом данных BOOLEAN. Возвращаемое значение по умолчанию установлено в FALSE, реализуя защитную стратегию. MmIsAddressValid() обычно вызывается непосредственно перед обращением к памяти по переданному ей адресу, чтобы избежать потенциальной ошибки страницы. Поэтому если бы функция возвращала TRUE в случае невозможности получения действительного результата из-за ошибки IOCTL, это повысило бы риск возникновения BSOD.

Это было несложно. Давайте теперь рассмотрим, как в этой среде можно организовать доступ к экспортируемым переменным. В листинге 6.25 показаны две функции-посредника, _NtBuildNumber() и _KeServiceDescriptorTable(). NtBuildNumber() и экспортируется модулем ntoskrnl.exe, как 16-битная переменная типа WORD, поэтому ей соответствует функция w2kCopy16() из библиотеки w2kcall.dll. В случае ошибки функция-посредник возвращает ноль (если у вас есть идея получше, дайте мне знать, пожалуйста). Функция-посредник _KeServiceDescriptorTable() реализована несколько по-другому, поскольку исходный экспортируемый ntoskrnl.exe адрес KeServiceDescriptorTable указывает на структуру, состоящую из более чем 64 бит. В этом случае лучше всего вернуть сам адрес KeServiceDescriptorTable, вместо того чтобы считывать ее неполные данные. Поэтому функция-посредник использует вспомогательную функцию w2kCopyEP() из листинга 6.23.

Листинг 6.25. Пример функции-посредника для переменных NtBuildNumber и KeServiceDescriptorTable

```
WORD WINAPI
_NtBuildNumber (VOID)
      {
      return w2kCopy16 (0, "NtBuildNumber");
```

```
    }
//  ------------------------------------------------------

PSERVICE_DESCRIPTOR_TABLE  WINAPI
_KeServiceDescriptorTable  (VOID)
    {
    return w2kCopyEP (NULL, "KeServiceDescriptorTable");
    }
```

Можете представить себе мою радость, когда я убедился в том, что эти функции-посредники действительно работают! Затем я подумал: попробую-ка я вызвать функции очень низкого уровня, работающие непосредственно с оборудованием, осуществляющие чтение и запись в порты ввода-вывода и т. п. К счастью, я спроектировал функцию SpyModuleSymbolEx() (листинг 6.11) таким образом, что поиск адресов по идентификаторам становится возможным в *любом* системном модуле, в том числе и в драйверах режима ядра. Моей следующей задачей было вызвать некоторые функции, экспортируемые уровнем аппаратных абстракций Windows 2000 (Hardware Abstraction Layer, HAL). Просмотрев список идентификаторов в разделе экспорта hal.dll, я решил попробовать написать две простые функции для обращения к оборудованию напрямую: HalMakeBeep() и HalQueryRealTimeClock(). Создание HalMakeBeep() напомнило мне старые добрые времена DOS, когда можно было заставить динамик компьютера пищать разнообразными способами, программируя аппаратные микросхемы на материнской плате. Фактически реализация HalMakeBeep() очень похожа на мои старые ассемблерные программы 1987 г., которые проигрывали целые мелодии по заданному массиву высот тона и периодов звучания. При работе с динамиком компьютера нужно программировать таймер и микросхему параллельного ввода-вывода (parallel I/O, PIO) по адресам ввода-вывода 0x0042, 0x0043 и 0x0061, поэтому функция HalMakeBeep() идеально подходила на роль первой тестовой функции-посредника для зависимой от оборудования функции, гарантируя при этом немедленную воспринимаемую на слух обратную связь.

В листинге 6.26 показана реализация функции-посредника _HalMakeBeep(), исключительно простой фрагмент кода, что стало возможным благодаря вспомогательной функции w2kCall08(). HalMakeBeep() начинает с гудка динамика с запрашиваемой высотой тона. Если задающий высоту аргумент установить в ноль, гудок прерывается. Функция возвращает TRUE при допустимом значении высоты, то есть ноль или больше 18. Заметьте, что определяющая идентификатор символьная строка в функции w2kCall08() включает имя требуемого модуля, hal.dll в нашем случае. В листингах 6.24 и 6.25 имя модуля не указывалось, поскольку рассматриваемые там идентификаторы экспортируются модулем по умолчанию ntoskrnl.exe.

Хотя HalMakeBeep() весьма тривиальная функция, я был очень счастлив, увидев, что _HalMakeBeep() работает. Динамик компьютера звучал по моему запросу! И это делала Windows 2000, а не DOS, служа доказательством тому, что приложение Win32 может вызывать функцию HAL, которая обращается непосредственно к оборудованию. Я перенес мой старый синтезатор из DOS в Windows 2000, итоговый код показан в листинге 6.27. w2kBeep() издает один звук с заданной высотой тона и длительностью. w2kBeepEx() последовательно проигрывает значения высот/периодов звучания из массива, пока не прочтет значение с нулевым периодом звучания. Обе функции экспортируются hal.dll. Может быть, с их помощью вы добави-

те в свои приложения Win32 музыкальное сопровождение, которое даст ощущение классической программы под DOS.

Листинг 6.26. Посредник для функции нижнего уровня HalMakeBeep()

```
BOOLEAN WINAPI
_HalMakeBeep (DWORD Pitch)
      {
      return w2kCall108 (FALSE, "hal.dll!HalMakeBeep", FALSE,
                         4, Pitch);
      }
```

Листинг 6.27. Простой синтезатор

```
BOOL WINAPI
w2kBeep (DWORD  dDuration,
         DWORD  dPitch)
      {
      BOOL fOk = TRUE;

      if (!_HalMakeBeep (dPitch)) fOk = FALSE;
      Sleep (dDuration);
      if (!_HalMakeBeep (0      )) fOk = FALSE;
      return fOk;
      }

//  ------------------------------------------------------

BOOL WINAPI
w2kBeepEx (DWORD dData,
           ...)
      {
      PDWORD    pdData;
      BOOL fOk = TRUE;

      for (pdData = &dData; pdData [0]; pdData += 2)
           {
           if (!w2kBeep (pdData [0], pdData [1])) fOk = FALSE;
           }
      return fOk;
      }
```

Затем я попытался написать посредника для более полезной функции, такой как HalQueryRealTimeClock(). Я помню, что обращение к часам реального времени на плате из приложений DOS одно время считалось трудным. При этом было необходимо осуществлять чтение и запись в несколько аппаратных портов ввода-вывода. В листинге 6.28 показаны функции-посредники для HalQueryRealTimeClock(), а также структура _TIME_FIELDS (определенная в файле ntddk.h), с которой работают обе функции.

Листинг 6.28. Функции-посредники для HalQueryRealTimeClock() и HalSetQueryRealTimeClock()

```
typedef struct _TIME_FIELDS
      {
      SHORT Year;
      SHORT Month;
      SHORT Day;
```

```
      SHORT Hour;
      SHORT Minute.
      SHORT Second.
      SHORT Milliseconds;
      SHORT Weekday; // 0 = воскресенье
      }
      TIME_FIELDS. * PTIME_FIELDS;

//  -------------------------------------------------------

#define TIME_FIELDS_ \
               sizeof (TIME_FIELDS)

VOID WINAPI
_HalQueryRealTimeClock (PTIME_FIELDS TimeFields)
      {
      w2kCallV (NULL, "hal.dll!HalQueryRealTimeClock",
               FALSE, 4, TimeFields);
      return;
      }

//  -------------------------------------------------------

VOID WINAPI
_HalSetRealTimeClock (PTIME_FIELDS TimeFields)
      {
      w2kCallV (NULL, "hal.dll!HalSetRealTimeClock", FALSE,
               4, TimeFields);
      return;
      }
```

В листинге 6.29 показан типичный способ применения функции _HalSetRealTimeClock(), отображающий текущие дату и время в окне консоли.

Листинг 6.29. Отображение текущих даты и времени

```
VOID WINAPI DisplayTime (void)
      {
      TIME_FIELDS tf;

      HalQueryRealTimeClock (&tf);

               printf (
L"%\r\nDate\Time: %02hd-%02hd-%04hd %02hd:%02hd:%02hd\r\n",
               tf.Month, tf.Day,       tf.Year,
               tf.Hour, tf.Minute,     tf.Second);
      return;
      }
```

Хотя работоспособность интерфейса вызовов ядра — замечательный факт, в то же время это несколько тревожно. Нас долгие годы учили, что Windows NT/2000 — это безопасная операционная система, в которой приложению не разрешается делать все что ему вздумается. Средний программист Win32 был отрезан от оборудования. Более опытный программист для NT по крайней мере знал, как вызывать функции Native API через ntdll.dll. Эксперт программирования в среде NT мог писать драйверы режима ядра для выполнения операций, не разрешенных в пользова-

тельском режиме. Теперь же при помощи представленной здесь DLL всем программистам Win32 открыт доступ к вызову произвольных функций ядра, так же как к любой другой функции Win32 API. Что это — большая дыра в безопасности в ядре Windows 2000? Нет, это не так. Единственная безопасная на 100% система — это система, которая не предоставляет приложениям вообще никакой возможности доступа, а такая система, конечно, бесполезна. Как только появляется способ взаимодействия с системой, она становится уязвимой. И как только производитель операционной системы позволяет разработчикам третьих фирм добавлять в систему свои компоненты, становится возможным при помощи официально не признаваемых методов построить прямой мост к ядру, такой, например, как пара модулей w2k_spy.sys/w2k_call.dll. Если система взаимодействует с окружением, такого понятия, как 100%-ная безопасность системы, просто не существует.

Функции, обеспечивающие доступ к данным

В библиотеку w2k_call.dll я добавил несколько дюжин функций-посредников API ядра. Например, весь набор функций управления строками из библиотеки времени выполнения Windows 2000 доступен через эту DLL. Однако если вы поэкспериментируете с этими предопределенными или добавленными вами функциями-посредниками, вы обнаружите, что вызов функций API ядра из пользовательского режима слегка отличается от вызова обычных функций Win32. Простота представленного здесь интерфейса вызовов ядра немного скрывает тот факт, что вызывающее приложение все равно остается программой пользовательского режима с ограниченными привилегиями. Например, приложение может вызвать функцию ядра, возвращающую указатель на структуру UNICODE_STRING. Скорее всего, это будет указатель на область памяти режима ядра, которая невидима для вызывающего приложения. Любые попытки доступа к данным этой строки приведут к завершению приложения с возникновением исключения, сообщающего о том, что инструкция по такому-то адресу попыталась прочитать данные по запрещенному адресу. Чтобы решить эту проблему, я добавил в библиотеку w2k_call.dll функции поддержки, обеспечивающие простой доступ к самым распространенным типам данных, используемых при вызовах функций API.

Универсальная функция w2kSpyRead() из листинга 6.30 копирует произвольные блоки памяти в переданный вызывающей программой буфер. Она основывается на функции IOCTL SPY_IO_MEMORY_BLOCK из модуля w2k_spy.sys драйвера слежения, кратко описанной в главе 4. Применяйте эту функцию для чтения содержимого отдельных членов структур, размещенных в памяти ядра. Важно отметить, что функция w2kSpyRead() завершится неудачно, если диапазон адресов, занимаемый блоком памяти, содержит недопустимые адреса. «Недопустимый» (invalid) означает, что этому адресу не соответствует ни физическая память, ни память в файле подкачки. w2kSpyClone() — расширенная версия функции w2kSpyRead(), которая автоматически выделяет буфер нужного размера и копирует в него данные ядра.

Листинг 6.30. Универсальная функция доступа к данным

```
BOOL WINAPI w2kSpyRead (PVOID  pBuffer,
                        PVOID  pAddress,
                        DWORD  dBytes)
      {
      SPY_MEMORY_BLOCK  smb;
      BOOL              fOk = FALSE;

      if ((pBuffer != NULL) && (pAddress != NULL) && dBytes)
           {
           ZeroMemory (pBuffer, dBytes);

           smb.pAddress   = pAddress;
           smb.dBytes     = dBytes;

           fOk = w2kSpyControl (SPY_IO_MEMORY_BLOCK,
                                &smb,    SPY_MEMORY_BLOCK_,
                                pBuffer, dBytes);
           }
      return fOk;
      }

//  ------------------------------------------------------

PVOID WINAPI w2kSpyClone (PVOID  pAddress,
                          DWORD  dBytes)
      {
      PVOID pBuffer = NULL;

      if ((pAddress != NULL) && dBytes &&
           ((pBuffer = w2kMemoryCreate (dBytes)) != NULL) &&
           (!w2kSpyRead (pBuffer, pAddress, dBytes)))
           {
           pBuffer = w2kMemoryDestroy (pBuffer);
           }
      return pBuffer;
      }
```

Для чтения строк необходимо сделать несколько больше. Вспомните, что компоненты режима ядра чаще всего в качестве строкового типа используют структуру UNICODE_STRING, в которой содержится указатель на буфер строки, а также информация о размере буфера и количество занимаемых строкой байт. Чтение структуры UNICODE_STRING, как правило, происходит в два этапа. Сначала нужно скопировать структуру UNICODE_STRING, чтобы найти размер и адрес буфера строки. Затем читаются данные строки. Для упрощения часто выполняемой задачи w2k_call.dll предоставляет набор функций, приведенных в листинге 6.31. Функции w2kStringAnsi() и w2kStringUnicode() выделяют и инициализируют соответственно пустые структуры ANSI_STRING и UNICODE_STRING, в том числе и буфер строки заданного размера. Для простоты буфер и заголовок строки объединены в одном блоке памяти. В эти структуры можно копировать строки, как демонстрирует функция w2kStringClone(). Она создает точную копию строки UNICODE_STRING в памяти пользовательского режима. Значение MaximumLength копии обычно равно исходному, кроме того слу-

чая, когда параметры исходной строки некорректны. Например, если значение MaximumLength меньше или равно значению члена Length, значит, оно некорректно и поэтому устанавливается в Length+2. Тем не менее значение MaximumLength копии никогда не будет меньше, чем исходное значение MaximumLength.

Листинг 6.31. Функции управления строками

```
PANSI_STRING WINAPI w2kStringAnsi (DWORD dSize)
      {
      PANSI_STRING pasData = NULL;

      if ((pasData = w2kMemoryCreate (ANSI_STRING_ + dSize))
           != NULL)
           {
           pasData->Length         = 0,
           pasData->MaximumLength  = (WORD) dSize;
           pasData->Buffer         = PTR_ADD (pasData, ANSI_STRING_),

           if (dSize) pasData->Buffer [0] = 0;
           }
      return pasData.
      }

//  ------------------------------------------------------

PUNICODE_STRING WINAPI w2kStringUnicode (DWORD dSize)
      {
      DWORD              dSize1    = dSize * WORD_,
      PUNICODE_STRING    pusData   = NULL;

      if ((pusData = w2kMemoryCreate (UNICODE_STRING_
                                      + dSize1))
           != NULL)
           {
           pusData->Length         = 0,
           pusData->MaximumLength  = (WORD) dSize1,
           pusData->Buffer         = PTR_ADD (pusData,
                                              UNICODE_STRING_),

           if (dSize) pusData->Buffer [0] = 0,
           }
      return pusData;
      }

//  ------------------------------------------------------

PUNICODE_STRING WINAPI w2kStringClone (PUNICODE_STRING pusSource)
      {
      DWORD              dSize;
      UNICODE_STRING     usCopy;
      PUNICODE_STRING    pusData = NULL,

      if (w2kSpyRead (&usCopy, pusSource, UNICODE_STRING_))
           {
           dSize = max (usCopy.Length + WORD_,
                        usCopy.MaximumLength) / WORD_;
```

```
        if (((pusData = w2kStringUnicode (dSize)) != NULL)
            &&
            usCopy.Length && (usCopy.Buffer != NULL))
            {
            if (w2kSpyRead (pusData->Buffer, usCopy.Buffer,
                            usCopy.Length))
                {
                pusData->Length = usCopy.Length;
                pusData->Buffer [usCopy.Length / WORD_] = 0;
                }
            else
                {
                pusData = w2kMemoryDestroy (pusData);
                }
            }
        }
    return pusData;
    }
```

Можно также копировать строки в область памяти приложения при помощи одной из функций ядра времени выполнения. Например, того же результата можно достичь при помощи двух функций-посредников, _RtlInitUnicodeString() и _RtlCopyUnicodeString, из модуля w2k_call.dll. Но, как правило, проще вызывать w2kStringClone(), поскольку эта функция автоматически выделяет память для копии строки.

Доступ к неэкспортируемым идентификаторам

К этому моменту мы достигли следующего: приложение режима пользователя может выполнять действия, ранее предоставленные исключительно драйверам режима ядра. Можем ли мы расширить приложение, добавив ему способности, которыми не обладает даже драйвер режима ядра? Можем ли мы вызывать внутренние функции, которые не только не документированы, но и не экспортируются? Звучит рискованно, но, как я покажу в этом разделе, это не так страшно, как кажется, если проявлять должную осторожность.

Поиск внутренних идентификаторов

Интерфейс вызовов ядра, описанный в предыдущих разделах, предоставляет задачу поиска адресов экспортируемых идентификаторов драйверу слежения, который обладает полным доступом к образам PE модулей ядра, расположенных в старшей половине линейного адресного пространства. Тем не менее если вызываемая функция или глобальная переменная, к которой требуется доступ, не экспортируется, драйвер слежения никак не сможет найти ее адрес. При написании этой главы и изучении листингов дизассемблированного кода, полученного при помощи Kernel Debugger, я часто думал: «Как жаль, что они не стали экспортировать эту замечательную функцию!» Особенно меня раздражало то, что Kernel Debugger показывал точное имя функции, однако коду приложения оно было недоступно.

Конечно, я мог бы воспользоваться интерфейсом вызовов ядра для перехода на простую двоичную точку входа функции, но такой стиль программирования не очень хорош. При очередном пакете обновления эта точка входа может переместиться на совершенно другой адрес.

Я понял, что если это может делать Kernel Debugger, то и мое приложение тоже должно быть на это способно. Пример DLL из главы 1 вывел меня на правильный путь. Библиотека w2k_img.dll предоставляет все необходимое для поиска адреса любого идентификатора, определенного модулями ядра Windows 2000, если файлы идентификаторов операционной системы установлены правильно. Я добавил в w2k_call.dll функцию API, которая получает линейный адрес идентификатора по его символьному имени и затем обращается по этому адресу при помощи w2kCall(). Разумеется, такая же функция есть и для глобальных переменных.

В листинге 6.32 показан полный набор расширенного интерфейса вызовов. Как и раньше, для всех основных типов функций предоставляется отдельная удобная функция, соответствующая одной из функций в листингах с 6.20 по 6.22. Основную работу делает функция w2kXCall(). Для получения адреса переданного ей идентификатора она вызывает функцию API imgTableResolve() из библиотеки w2k_img.dll и в случае успешного определения адреса передает его при последовательных вызовах w2kCall(). Так как происходит обращение по адресу, а не вызов идентификатора, параметру pbSymbol функции w2kCall() передается NULL-указатель. Аргумент pEntryPoint установлен в значение адреса идентификатора pie->pAddress, только что полученное из файлов идентификаторов. Как объяснялось в главе 1, библиотека w2k_img.dll способна определить соглашения о передаче параметров большинства внутренних функций, поэтому аргумент fFastCall может быть установлен автоматически путем проверки, не принимает ли pie->dConvention значение CONVENTION_FASTCALL. Число байт аргументов и указатель на аргументы принимаются от вызывающей функции и передаются дальше. Можно было бы получать число аргументов также и из идентификаторов, но это работает только для функций __stdcall и __fastcall. Идентификаторы __cdecl при декодировании имени функции не включают в него информацию о размере стека аргументов.

Листинг 6.32. Расширенный интерфейс вызовов

```
BOOL WINAPI w2kXCall (PULARGE_INTEGER  puliResult,
                      PBYTE            pbSymbol,
                      DWORD            dArgumentBytes,
                      PVOID            pArguments)
    {
    PIMG_TABLE     pit;
    PIMG_ENTRY     pie;
    BOOL           fOk = FALSE;

    if (((pit = w2kSymbolsGlobal (NULL))           != NULL)
        &&
        ((pie = imgTableResolve (pit, pbSymbol))   != NULL)
        &&
        (pie->pAddress != NULL))
        {
        fOk = w2kCall (puliResult, NULL, pie->pAddress,
```

```
                    pie->dConvention == IMG_CONVENTION_FASTCALL,
                    dArgumentBytes, pArguments);
        }
    else
        {
        if (puliResult != NULL) puliResult->QuadPart = 0;
        }
    return fOk;
    }

//  ------------------------------------------------------

BOOL WINAPI w2kXCallV (PULARGE_INTEGER  puliResult,
                       PBYTE            pbSymbol,
                       DWORD            dArgumentBytes,
                       ...)
    {
    return w2kXCall (puliResult, pbSymbol,
                     dArgumentBytes, &dArgumentBytes + 1);
    }

//  ------------------------------------------------------

NTSTATUS WINAPI w2kXCallNT (PBYTE pbSymbol,
                            DWORD dArgumentBytes,
                            ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR);
    }

//  ------------------------------------------------------

BYTE WINAPI w2kXCall08 (BYTE   bDefault,
                        PBYTE  pbSymbol,
                        DWORD  dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (BYTE) uliResult.LowPart
            : bDefault);
    }

//  ------------------------------------------------------

WORD WINAPI w2kXCall16 (WORD   wDefault,
                        PBYTE  pbSymbol,
                        DWORD  dArgumentBytes,
                        ...)
```

продолжение ⇩

Листинг 6.32 (*продолжение*)

```
    {
    ULARGE_INTEGER ulResult.

    return (w2kXCall (&ulResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (WORD) ulResult.LowPart
            : wDefault);
    }

//  ------------------------------------------------------

DWORD WINAPI w2kXCall32 (DWORD  dDefault,
                         PBYTE  pbSymbol,
                         DWORD  dArgumentBytes,
                         ...)
    {
    ULARGE_INTEGER ulResult;

    return (w2kXCall (&ulResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? ulResult.LowPart
            : dDefault);
    }

//  ------------------------------------------------------

QWORD WINAPI w2kXCall64 (QWORD  qDefault,
                         PBYTE  pbSymbol,
                         DWORD  dArgumentBytes,
                         ...)
    {
    ULARGE_INTEGER ulResult;

    return (w2kXCall (&ulResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? ulResult.QuadPart
            : qDefault);
    }

//  ------------------------------------------------------

PVOID WINAPI w2kXCallP (PVOID pDefault,
                        PBYTE pbSymbol,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult.

    return (w2kXCall (&ulResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (PVDID) ulResult LowPart
            : pDefault);
    }
```

Заметьте, что в листинге 6.32 w2kXCall() в первую очередь вызывает w2kSymbolsGlobal(). Эта функция включена в листинг 6.33 вместе со вспомогательными функциями, она предназначена для того, чтобы загрузить таблицу идентификаторов ntoskrnl.exe при первом выполнении w2kXCall(). Таблица сохраняется в глобальной переменной gpit типа PING_TABLE, поэтому последующие вызовы смогут снова ей воспользоваться. При поддержке вспомогательных функций w2kSymbolsLoad() возвращает один из кодов состояния, перечисленных в табл. 6.2 в необязательном аргументе *pdStatus. Чтобы не допускать перехода на недопустимый адрес из-за не найденного для идентификатора адреса, w2kSymbolsLoad() предусмотрительно сверяет временную метку и контрольную сумму файлов идентификаторов с соответствующими полями в постоянно находящемся в памяти образе требуемого модуля при помощи функции w2kPeCheck() (в книге не приведена) и отбрасывает таблицу идентификаторов, если они в точности не совпадают.

Листинг 6.33. Функции диспетчера таблицы идентификаторов

```
PIMG_TABLE WINAPI w2kSymbolsLoad (PBYTE   pbModule,
                                  PDWORD  pdStatus)
     {
     PVOID         pBase;
     DWORD         dStatus   = W2K_SYMBOLS_UNDEFINED;
     PIMG_TABLE    pit       = NULL;

     if ((pBase = imgModuleBaseA (pbModule)) == NULL)
          {
          dStatus = W2K_SYMBOLS_MODULE_ERROR;
          }
     else
          {
          if ((pit = imgTableLoadA (pbModule, pBase))
               == NULL)
               {
               dStatus = W2K_SYMBOLS_LOAD_ERRDR;
               }
          else
               {
               if (!w2kPeCheck (pbModule, pit->dTimeStamp,
                                pit->dCheckSum))
                    {
                    dStatus   = W2K_SYMBOLS_VERSION_ERROR;
                    pit       = imgMemoryDestroy (pit);
                    }
               else
                    {
                    dStatus = W2K_SYMBOLS_OK;
                    }
               }
          }
     if (pdStatus != NULL) *pdStatus = dStatus;
     return pit;
     }
```

продолжение ⇩

Листинг 6.33 (*продолжение*)

```
//  -----------------------------------------------------

PIMG_TABLE WINAPI w2kSymbolsGlobal (PDWORD pdStatus)
      {
      DWORD          dStatus   = W2K_SYMBOLS_UNDEFINED;
      PIMG_TABLE     pit       = NULL.

      w2kSpyLock ().

      if ((gdStatus == W2K_SYMBOLS_OK) && (gpit == NULL))
           {
           gpit = w2kSymbolsLoad (NULL, &gdStatus).
           }
      dStatus = gdStatus.
      pit     = gpit:

      w2kSpyUnlock ().

      if (pdStatus != NULL) *pdStatus = dStatus.
      return pit.
      }

//  -----------------------------------------------------

DWORD WINAPI w2kSymbolsStatus (VOID)
      {
      DWORD dStatus = W2K_SYMBOLS_UNDEFINED.

      w2kSymbolsGlobal (&dStatus):
      return dStatus.
      }

//  -----------------------------------------------------

VOID WINAPI w2kSymbolsReset (VOID)
      {
      w2kSpyLock ():

      gpit      = imgMemoryDestroy (gpit).
      gdStatus  = W2K_SYMBOLS_OK.

      w2kSpyUnlock ().
      return.
      }
```

Функции w2kSymbolsStatus() и w2kSymbolsReset() в нижней части листинга 6.33 применяются для загрузки и выгрузки таблицы идентификаторов по запросу. w2kSymbolsStatus() пытается загрузить таблицу идентификаторов, если она еще не загружена, и возвращает ее состояние. Если w2k_call.dll уже пыталась загрузить таблицу и ей это не удалось, функция просто возвращает код последней ошибки (табл. 6.2), пока таблица идентификаторов не будет переустановлена вызовом w2kSymbolsReset(). Последняя функция также очищает блок памяти, занимаемый

таблицей идентификаторов (если таковой существует), что обеспечит полную перезагрузку идентификаторов при следующем запросе, в котором участвует таблица идентификаторов ntoskrnl.exe.

Таблица 6.2. Коды возврата функции w2kSymbolsLoad()

Код возврата	Описание
W2K_SYMBOLS_OK	Таблица идентификаторов модуля была загружена
W2K_SYMBOLS_MODULE_ERROR	Модуль не находится в памяти
W2K_SYMBOLS_LOAD_ERROR	Не удается загрузить файлы идентификаторов модуля
W2K_SYMBOLS_VERSION_ERROR	Файлы идентификаторов не соответствуют образу модуля в памяти
W2K_SYMBOLS_UNDEFINED	Состояние таблицы идентификаторов не определено

Набор функций w2kXCopy*(), реализующих расширенный интерфейс копирования, показан в листинге 6.34, который соответствует приведенному выше листингу 6.23. w2kXCopy() просто вызывает w2kXCall() с отрицательным значением параметра dArgumentBytes, остальные функции копирования представляют собой просто оболочки функции w2kXCopy() с упрощенными списками параметров.

Листинг 6.34. Расширенный интерфейс копирования

```
BOOL WINAPI w2kXCopy (PULARGE_INTEGER  puliResult,
                      PBYTE            pbSymbol
                      DWORD            dBytes)
      {
      return w2kXCall (puliResult, pbSymbol,
                       0xFFFFFFFF - dBytes, NULL);
      }

//  ------------------------------------------------------

BYTE WINAPI w2kXCopyOB (BYTE   bDefault,
                        PBYTE  pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kXCopy (&uliResult, pbSymbol, 1)
               ? (BYTE) uliResult.LowPart
                 bDefault);
      }

//  ------------------------------------------------------

WORD WINAPI w2kXCopy16 (WORD   wDefault,
                        PBYTE  pbSymbol)
      {
      ULARGE_INTEGER uliResult;

      return (w2kXCopy (&uliResult, pbSymbol, 2)
               ? (WORD) uliResult.LowPart
               : wDefault);
      }
```

продолжение

Листинг 6.34 (*продолжение*)

```
//  ------------------------------------------------------

DWORD WINAPI w2kXCopy32 (DWORD  dDefault
                         PBYTE  pbSymbol)
      {
      ULARGE_INTEGER ulResult;

      return (w2kXCopy (&ulResult, pbSymbol, 4)
               ? ulResult LowPart
               : dDefault);
      }

//  ------------------------------------------------------

QWORD WINAPI w2kXCopy64 (QWORD qDefault,
                         PBYTE pbSymbol)
      {
      ULARGE_INTEGER ulResult;

      return (w2kXCopy (&ulResult, pbSymbol, 8)
               ? ulResult QuadPart
                 qDefault);
      }

//  ------------------------------------------------------

PVOID WINAPI w2kXCopyP (PVOID pDefault,
                        PBYTE pbSymbol)
      {
      ULARGE_INTEGER ulResult;

      return (w2kXCopy (&ulResult, pbSymbol, 4)
               ? (PVOID) ulResult LowPart
                 pDefault);
      }

//  ------------------------------------------------------

PVOID WINAPI w2kXCopyEP (PVOID pDefault,
                         PBYTE pbSymbol)
      {
      ULARGE_INTEGER ulResult;

      return (w2kXCopy (&ulResult, pbSymbol, 0)
               ? (PVOID) ulResult.LowPart
               . pDefault);
      }
```

Реализация посредников для функций ядра

При реализации посредников для внутренних функций ядра годятся те же предпосылки, что и для экспортируемых функций API, за тем исключением, что могут быть вызваны только функции из модуля ntoskrnl.exe. Такое ограничение накла-

дывает диспетчер таблицы идентификаторов в библиотеке w2k_call.dll, а не сам интерфейс вызовов. Для простоты загружается только таблица идентификаторов ntoskrnl.exe, потому что именно в этом модуле находятся наиболее интересные идентификаторы (разумеется, библиотеку w2k_call.dll можно было расширить так, чтобы по запросу загружалось несколько таблиц). В листинге 6.35 приведены два примера посредников для внутренних функций диспетчера объектов Windows 2000, возвращающих информацию о типах объектов (типы объектов будут подробно обсуждены в главе 7).

В листинге 6.36 показаны три функции-посредника для очень важных внутренних структур данных, которые будут затем использоваться в программе-примере в главе 7. Обратите внимание: к именам всех функций-посредников, обращающихся к расширенному интерфейсу вызовов ядра, я добавил префикс из двух символов подчеркивания. Это сделано как напоминание о том, что функция будет работать только с надлежащим набором файлов идентификаторов. Если вы установите пакет обновления, не обновив при этом файлы идентификаторов, функция w2kSymbolsLoad() не сможет загрузить идентификаторы и функции-посредники завершатся неудачно и вернут значения по умолчанию. С другой стороны, функции-посредники с префиксом из одного символа подчеркивания будут работать с несоответствующими файлами идентификаторов, потому что они будут искать адреса для идентификаторов на основе постоянно находящихся в памяти таблиц экспорта новых модулей. Несмотря на это, после обновления они также могут завершиться неудачно, если обновленные модули будут экспортировать не все требуемые функции API или у каких-либо функций изменятся списки параметров.

Листинг 6.35. Примеры функций-посредников для ObQueryTypeInfo() и ObQueryTypeName()

```
NTSTATUS WINAPI
__ObQueryTypeInfo (  POBJECT_TYPE        ObjectType,
                     POBJECT_TYPE_INFO   TypeInfo
                     DWORD               TypeInfoLength,
/* установить в 0 */ PDWORD             ReturnLength)
      {
      return w2kXCallNT ("ObQueryTypeInfo"
                         16, ObjectType, TypeInfo,
                         TypeInfoLength, ReturnLength),
      }

//  ------------------------------------------------------

NTSTATUS WINAPI
__ObQueryTypeName (POBJECT                    Object,
                   POBJECT_NAME_INFORMATION   NameString,
        /* байт */ DWORD                     NameStringLength,
                   PDWORD                     ReturnLength)
      {
      return w2kXCallNT ("ObQueryTypeName",
                         16  Object, NameString,
                         NameStringLength, ReturnLength),
      }
```

Листинг 6.36. Пример функций-посредников для переменных

```
PERESOURCE WINAPI
__ObpRootDirectoryMutex (VOID)
      {
      return w2kXCopyP (NULL, "ObpRootDirectoryMutex");
      }

//  ------------------------------------------------------

POBJECT_DIRECTORY WINAPI
__ObpRootDirectoryObject (VOID)
      {
      return w2kXCopyP (NULL, "ObpRootDirectoryObject");
      }

//  ------------------------------------------------------

POBJECT_DIRECTORY WINAPI
__ObpTypeDirectoryObject (VOID)
      {
      return w2kXCopyP (NULL, "ObpTypeDirectoryObject");
      }
```

На данный момент этого достаточно. Возможно, вы разочарованы отсутствием в этой главе кода, демонстрирующего использование функций API из w2k_call.dll, — не волнуйтесь, вы *получите* образец кода в следующей главе.

Управление объектами Windows 2000

Наверное, сложно найти внутри Windows 2000 нечто более впечатляющее, чем мир объектов этой операционной системы. Если пространство памяти, принадлежащее ОС, — это поверхность планеты, то объекты — это существа, обитающие на этой планете. Существует множество разновидностей объектов. Одни из них небольшие, другие — крупные, одни обладают простой структурой, другие чрезвычайно сложны. Все эти объекты взаимодействуют между собой самыми причудливыми способами. Операционная система Windows 2000 обладает продуманным, хорошо структурированным механизмом управления объектами, который фактически никак не документирован. В данной главе будет предпринята попытка представить на суд читателя краткий поверхностный обзор необъятной вселенной объектов Windows 2000. К сожалению, эта часть Windows 2000 является наиболее рьяно охраняемым секретом компании Microsoft, поэтому многие вопросы, поставленные в данной главе, оставлены мной без ответа. Однако я надеюсь на то, что данная глава будет неплохой отправной точкой для тех, кто осмелится шагнуть дальше меня в глубины Windows 2000, туда, «где еще не ступала нога человека».

Структуры объектов Windows 2000

В каталоге \src\common\include прилагаемого к данной книге компакт-диска содержится большой заголовочный файл с именем w2k_def.h, содержимое которого без сомнения заставит учащенно биться сердце любого системного программиста Windows 2000. В этом файле содержится большая коллекция констант и определений типов, собранная мною на протяжении многих лет исследований глубин Windows 2000/NT. Файл w2k_def.h можно использовать как при разработке приложений Win32, так и при создании драйверов режима ядра. Чтобы учесть различия этих двух сред компиляции, в файле используется компиляция по условию. Например, при разработке приложений Win32 нельзя воспользоваться заголо-

вочными файлами ntdef.h и ntddk.h, в которых содержится большая часть определений типов данных ядра. Чтобы решить эту проблему, все необходимые для работы с ядром операторы #define и typedef, которые можно обнаружить в заголовочных файлах пакета Device Development Kit (DDK) и которые требуются для определения недокументированных структур Windows 2000, включены в состав файла w2k_def.h. Чтобы избежать ошибок, вызванных повторным определением, все эти операторы размещаются в разделе #ifdef _USER_MODE_. Таким образом, если символ _USER_MODE_ не определен, компилятор игнорирует любые содержащиеся в данном разделе определения. Если вы разрабатываете приложение или динамическую библиотеку Win32 и при этом планируете использовать определения DDK, перед тем как включить в исходный код вашего приложения заголовочный файл w2k_def.h, добавьте в программу оператор #define _USER_MODE_. В разделе #else конструкции #ifdef _USER_MODE_ содержится небольшое количество полезных определений, отсутствующих в заголовочных файлах Windows 2000 DDK. Здесь можно обнаружить такие определения, как, например, SECURITY_DESCRIPTOR и SECURITY_DESCRIPTOR_CONTROL.

Основные категории объектов

Несмотря на то что объекты являются основой, на которой базируется вся Windows 2000, если вы попытаетесь обнаружить какую-либо связанную с ними полезную информацию в DDK, вы будете разочарованы: документация, описывающая структуру объектов и методы работы с ними, фактически отсутствует. Из двадцати одной функции API диспетчера объектов (их имена начинаются с суффикса Ob), экспортируемых модулем ntoskernl.exe, в документации DDK упоминаются только лишь шесть. Функции API, в качестве аргументов принимающие указатели на объекты, как правило, определяют эти указатели как переменные типа PVOID. Если вы попытаетесь найти что-либо, имеющее отношение к объектам в основных заголовочных файлах DDK с именами ntdef.h и ntddk.h, вы не обнаружите ничего интересного. Некоторые важные типы данных, имеющие отношение к внутренним объектам Windows 2000, определяются при помощи тривиальных операторов, добавленных в заголовочные файлы лишь для того, чтобы избежать ошибки компилятора. Например, структура OBJECT_TYPE определяется при помощи выражения typedef struct _OBJECT_TYPE *POBJECT_TYPE; Благодаря наличию такого определения компилятор сможет без проблем создать исполняемый файл, однако данное выражение не дает программисту ни малейшего представления о внутренней структуре определяемого типа данных.

Попытаемся ответить на вопрос: каково внутреннее устройство объектов Windows 2000? Если посмотреть на область памяти, на которую указывает указатель на объект, можно обнаружить, что соответствующий указателю линейный адрес делит расположенную в памяти структуру объекта на две части: *заголовок объекта* (object header) и *тело объекта* (object body). Другими словами, указатель на объект указывает не на самую первую ячейку памяти, принадлежащую структуре объекта, а на раздел объекта, содержащий тело этого объекта. Тело объекта расположе-

но непосредственно за заголовком объекта. Чтобы получить доступ к заголовку объекта, необходимо из значения указателя на объект вычесть некоторое обратное смещение. Внутреннее содержимое тела объекта целиком и полностью определяется типом объекта и для объектов разных типов строение тела может существенно отличаться. Наиболее простым объектом является объект события, его тело занимает в памяти всего 16 байт. К самым сложным относятся объекты потоков и процессов, для хранения тел этих объектов требуется по несколько сотен байт. Типы тел объектов можно разделить на три основные категории:

1. **Объекты синхронизации (dispatcher objects).** Эти объекты находятся на самом низком уровне системы. В начале тела каждого такого объекта располагается структура DISPATCHER_HEADER, которая для всех объектов синхронизации определяется одинаковым образом (листинг 7.1). В составе структуры хранится идентификатор типа объекта, а также размер тела объекта, измеренный в 32-битных единицах типа DWORD. Имена структур любых объектов синхронизации начинаются с буквы K, что означает *kernel — ядро*. Наличие заголовка DISPATCHER_HEADER означает, что объект является «ожидаемым» (waitable). Это значит, что за состоянием объекта можно следить, то есть приложения могут ожидать изменения состояния объекта. А это, в свою очередь, означает, что объект можно передать в качестве аргумента функциям синхронизации KeWaitForSingleObject() и KeWaitForMultipleObjects(). На обращении к этим двум функциям основана работа известных функций WaitForSingleObject() и WaitForMultipleObjects(), входящих в состав стандартного интерфейса Win32 API.
2. **Структуры данных, относящиеся к системе ввода/вывода (I/O system data structures).** Объекты данной категории иногда называют объектами ввода/вывода. Это высокоуровневые объекты, тело которых начинается с переменной типа SHORT, в которой хранится идентификатор типа объекта. Как правило, за этим идентификатором следует переменная типа SHORT или WORD, в которой хранится размер тела объекта, измеренный в 8-битных единицах типа BYTE. Однако не все объекты этой категории следуют данному правилу.
3. **Другие объекты.** К этой категории относится несколько объектов, которые нельзя отнести к первым двум описанным здесь категориям.

Листинг 7.1. Определение структуры DISPATCHER_HEADER

```
typedef struct _DISPATCHER_HEADER
        {
/*000*/    BYTE          Type;          // одно из значений DISP_TYPE_*
/*001*/    BYTE          Absolute,
/*002*/    BYTE          Size,          // количество элементов DWORD
/*003*/    BYTE          Inserted,
/*004*/    LONG          SignalState,
/*008*/    LIST_ENTRY    WaitListHead;
/*010*/    }
     DISPATCHER_HEADER,
 *   PDISPATCHER_HEADER,
 **  PPDISPATCHER_HEADER;
```

Обратите внимание на то, что идентификаторы назначаются объектам синхронизации независимо от идентификаторов, назначаемых структурам данных, относящимся к системе ввода/вывода (в дальнейшем я буду называть эти структуры объектами ввода/вывода), другими словами, пространства идентификаторов этих двух категорий объектов пересекаются между собой. В табл. 7.1 перечислены типы объектов синхронизации, о которых мне известно на текущий момент. Некоторые из структур, указанных в колонке «Структура С», определены в заголовочном файле ntddk.h пакета DDK. К сожалению, определения наиболее интересных из упоминаемых здесь структур, таких как, например, KPROCESS и KTHREAD, в DDK отсутствуют. Однако расстраиваться не стоит — позже в данной главе мы подробно рассмотрим эти специальные типы объектов. Определения любых недокументированных структур, строение которых известно мне хотя бы частично, содержатся в заголовочном файле w2k_def.h на прилагаемом к книге компакт-диске. Кроме того, вы можете найти их в приложении В данной книги.

Таблица 7.1. Перечень объектов синхронизации

ID	Тип	Структура С	Определеляется в файле
0	DISP_TYPE_NOTIFICATION_EVENT	KEVENT	ntddk.h
1	DISP_TYPE_SYNCHRONIZATION_EVENT	KEVENT	ntddk.h
2	DISP_TYPE_MUTANT	KMUTANT, KMUTEX	ntddk.h
3	DISP_TYPE_PROCESS	KPROCESS	w2k_def.h
4	DISP_TYPE_QUEUE	KQUEUE	w2k_def.h
5	DISP_TYPE_SEMAPHORE	KSEMAPHORE	ntddk.h
6	DISP_TYPE_THREAD	KTHREAD	w2k_def.h
8	DISP_TYPE_NOTIFICATION_TIMER	KTIMER	ntddk.h
9	DISP_TYPE_SYNCHRONIZATION_TIMER	KTIMER	ntddk.h

В табл. 7.2 перечислены объекты ввода/вывода, о которых мне известно на текущий момент. В файле ntddk.h из пакета DDK определено лишь 13 первых идентификаторов. И снова в составе пакета DDK можно обнаружить описания лишь некоторых из структур, упомянутых в колонке «Структура С». Некоторые из структур, определения которых отсутствуют в DDK, определены в файле w2k_def.h, а также в приложении В данной книги.

Таблица 7.2. Перечень объектов ввода/вывода

ID	Тип	Структура С	Определеляется в файле
1	IO_TYPE_ADAPTER	ADAPTER_OBJECT	
2	IO_TYPE_CONTROLLER	CONTROLLER_OBJECT	ntddk.h
3	IO_TYPE_DEVICE	DEVICE_OBJECT	ntddk.h
4	IO_TYPE_DRIVER	DRIVER_OBJECT	ntddk.h
5	IO_TYPE_FILE	FILE_OBJECT	ntddk.h
6	IO_TYPE_IRP	IRP	ntddk.h
7	IO_TYPE_MASTER_ADAPTER		
8	IO_TYPE_OPEN_PACKET		
9	IO_TYPE_TIMER	IO_TIMER	w2k_def.h

ID	Тип	Структура C	Определяется в файле
10	IO_TYPE_VPB	VPB	ntddk.h
11	IO_TYPE_ERROR_LOG	IO_ERROR_LOG_ENTRY	w2k_def.h
12	IO_TYPE_ERROR_MESSAGE	IO_ERROR_LOG_MESSAGE	ntddk.h
13	IO_TYPE_DEVICE_OBJECT_EXTENTION	DEVOBJ_EXTENTION	ntddk.h
18	IO_TYPE_APC	KAPC	ntddk.h
19	IO_TYPE_DPC	KDPC	ntddk.h
20	IO_TYPE_DEVICE_QUEUE	KDEVICE_QUEUE	ntddk.h
21	IO_TYPE_EVENT_PAIR	KEVENT_PAIR	w2k_def.h
22	IO_TYPE_INTERRUPT	KINTERRUPT	
23	IO_TYPE_PROFILE	KPROFILE	

Заголовок объекта

Тело объекта может обладать любым внутренним строением, удобным для хранения сведений об объектах соответствующего типа. Диспетчер объектов Windows 2000 не накладывает никаких ограничений на размеры и внутреннюю структуру тела объекта. В противоположность этому строение заголовка объектов любых типов подчиняется строгим правилам. На рис. 7.1 показано строение объекта, заголовок которого обладает всеми возможными компонентами. В заголовок объекта любого типа входит по крайней мере одна базовая структура OBJECT_HEADER. В линейном адресном пространстве памяти эта структура располагается непосредственно перед телом объекта. Перед этой структурой могут располагаться еще до четырех дополнительных структур, содержащих разнообразную дополнительную информацию об объекте. Как уже упоминалось, указатель на объект содержит адрес первой ячейки тела объекта, таким образом чтобы получить доступ к структурам заголовка объекта, необходимо вычесть из указателя на объект некоторое смещение. Базовая структура заголовка объекта OBJECT_HEADER содержит информацию о том, какие дополнительные структуры входят в состав заголовка объекта и в каких местах заголовка они располагаются. Если в состав заголовка входят какие-либо дополнительные структуры помимо базовой, они располагаются в памяти перед базовой структурой одна за другой в порядке, указанном на рис. 7.1. Однако не стоит считать, что показанная на рисунке последовательность дополнительных структур одинакова абсолютно для всех объектов. Разрабатывая собственную программу, ни в коем случае не следует рассчитывать на то, что дополнительные структуры будут располагаться в заголовке объекта именно так, как это показано. Чтобы получить доступ к дополнительным структурам заголовка, необходимо проанализировать сведения, содержащиеся в базовой структуре OBJECT_HEADER. Эта структура содержит в себе все необходимое для того, чтобы обратиться к любой из дополнительных структур заголовка вне зависимости от того, как именно эти структуры размещаются в памяти. Подробнее об этом будет рассказано в самое ближайшее время. Сейчас же отмечу только то, что если в заголовке объекта присутствует структура OBJECT_CREATOR_INFO, то она обязательно располагается непосредственно перед структурой OBJECT_HEADER. Местоположение остальных дополнительных структур в заголовке не регламентировано.

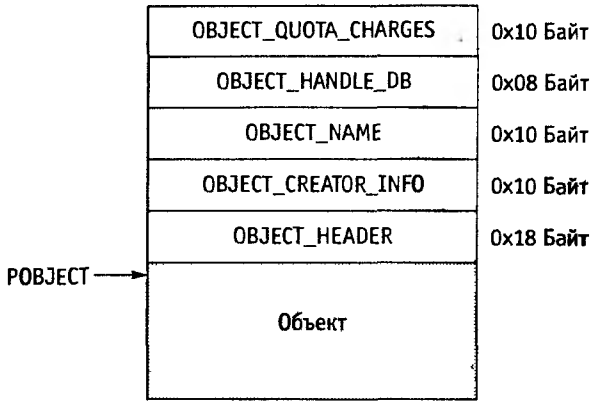

Рис. 7.1. Внутренняя структура объекта Windows 2000

Листинг 7.2. Структура OBJECT_HEADER

```
#define OB_FLAG_CREATE_INFO       0x01 // в заголовке есть раздел OBJECT_CREATE_INFO
#define OB_FLAG_KERNEL_MODE       0x02 // создан ядром
#define OB_FLAG_CREATOR_INFO      0x04 // в заголовке есть раздел OBJECT_CREATOR_INFO
#define OB_FLAG_EXCLUSIVE         0x08 // OBJ_EXCLUSIVE
#define OB_FLAG_PERMANENT         0x10 // OBJ_PERMANENT
#define OB_FLAG_SECURITY          0x20 // объект обладает дескриптором безопасности
#define OB_FLAG_SINGLE_PROCESS    0x40 // отсутствует список HandleOBList

typedef struct _OBJECT_HEADER
        {
/*000*/    DWORD          PointerCount;          // количество ссылок на объект
/*004*/    DWORD          HandleCount;           // количество открытых дескрипторов
/*008*/    POBJECT_TYPE   ObjectType;
/*00C*/    BYTE           NameOffset;            // -> OBJECT_NAME
/*00D*/    BYTE           HandleDBOffset;        // -> OBJECT_HANDLE_DB
/*00E*/    BYTE           QuotaChargesOffset;    // -> OBJECT_QUOTA_CHARGES
/*00F*/    BYTE           ObjectFlags;           // OB_FLAG_*
/*010*/    union
             { // OB_FLAG_CREATE_INFO ? ObjectCreateInfo : QuotaBlock
/*010*/        PQUOTA_BLOCK           QuotaBlock;
/*010*/        POBJECT_CREATE_INFO    ObjectCreateInfo;
/*014*/        };
/*014*/    PSECURITY_DESCRIPTOR  SecurityDescriptor;
/*018*/    }
     OBJECT_HEADER,
 *   POBJECT_HEADER,
 **  PPOBJECT_HEADER;
```

Определение структуры OBJECT_HEADER показано в листинге 7.2. Поля этой структуры имеют следующий смысл:

- PointerCount — указывает на то, какое количество ссылок на данный объект существует в настоящее время. Данное значение является аналогом счетчика обращений, который поддерживается для каждого объекта COM (Component Object Model). Функции ObfReferenceObject(), ObReferenceObjectByHandle(), ObReferenceObjectByName() и ObReferenceObjectByPointer(), входящие в состав

ntoskernl.exe, увеличивают значение PointerCount, в то время, как функции ObfReferenceObject() и ObDereferenceObject() уменьшают значение этого счетчика.

- HandleCount — это значение соответствует количеству открытых дескрипторов (handle), ссылающихся на данный объект.
- ObjectType — здесь содержится указатель на структуру OBJECT_TYPE, которая описывает объект типа, использовавшийся при создании данного объекта. Подробнее о структуре OBJECT_TYPE будет рассказано позже.
- NameOffset — количество байт, которое необходимо вычесть из адреса OBJECT_HEADER для того, чтобы получить адрес структуры OBJECT_NAME. Если данное значение равно нулю, значит, структура OBJECT_NAME отсутствует в заголовке объекта.
- HandleDBOffset — количество байт, которое необходимо вычесть из адреса OBJECT_HEADER для того, чтобы получить адрес структуры OBJECT_HANDLE_DB. Если данное значение равно нулю, значит, структура OBJECT_HANDLE_DB отсутствует в заголовке объекта.
- QuotaChargesOffset — количество байт, которое необходимо вычесть из адреса OBJECT_HEADER для того, чтобы получить адрес раздела OBJECT_QUOTA_CHARGES заголовка объекта. Если данное значение равно нулю, значит, структура OBJECT_QUOTA_CHARGES отсутствует в заголовке объекта.
- ObjectFlags — в данном поле содержатся различные бинарные свойства объекта. Значения различных битов этого поля описываются в начале листинга 7.2. Если бит OB_FLAG_CREATOR_INFO установлен, значит, в заголовке объекта присутствует структура OBJECT_CREATOR_INFO, которая располагается непосредственно перед структурой OBJECT_HEADER. В книге «Windows NT/2000 Native API Reference» при описании класса SystemObjectInformation функции ZwQuerySystemInformation() Гари Неббетт (Gary Nebbett) обозначает эти флаги несколько отличающимися именами (Nebbet G. «Windows NT/2000 Native API Reference». Indianapolice, IN: Macmillan Technical Publishing MTP, 2000, с. 24), которые перечисляются в табл. 7.3.
- QuotaBlock и ObjectCreateInfo — эти поля взаимно исключают друг друга. Иными словами, структура OBJECT_HEADER не может содержать в себе обе этих переменных одновременно — если в структуре OBJECT_HEADER присутствует одна из них, значит, второй быть не должно. Выбор конкретной переменной определяется значением флага OB_FLAG_CREATE_INFO в поле ObjectFlags. Если этот флаг установлен, значит, в данной позиции структуры OBJECT_HEADER содержится переменная ObjectCreateInfo, которая является указателем, указывающим на структуру OBJECT_CREATE_INFO, использовавшуюся при создании данного объекта. Если же флаг OB_FLAG_CREATE_INFO сброшен, значит, в данной позиции содержится переменная QuotaBlock, которая является указателем на структуру QUOTA_BLOCK, в которой содержатся сведения об использовании свопируемой и несвопируемой памяти. Указатель QuotaBlock многих объектов указывает на внутреннюю структуру PspDefaultQuotaBlock. Значение данного объединения может быть также равно NULL.
- SecurityDescriptor — если флаг OB_FLAG_SECURITY поля ObjectFlags установлен, то в данной переменной содержится указатель на структуру SECURITY_ DESCRIPTOR. В противном случае значение данного поля равно NULL.

Таблица 7.3. Сравнение различных версий интерпретации флагов битовой маски ObjectFlags

Шребер (Schreiber)	Значение	Неббетт (Nebbett)
OB_FLAG_CREATE_INFO	0x01	отсутствует
OB_FLAG_KERNEL_MODE	0x02	KERNE
OB_FLAG_CREATOR_INFO	0x04	CREATOR_INFO
OB_FLAG_EXCLUSIVE	0x08	EXCLUSIVE
OB_FLAG_PERMANENT	0x10	PERMANENT
OB_FLAG_SECURITY	0x20	DEFAULT_SECURITY_QUOTA
OB_FLAG_SINGLE_PROCESS	0x40	SINGLE_HANDLE_ENTRY

Описывая поля структуры OBJECT_HEADER, я упомянул о нескольких важных структурах, внутреннее строение которых хотелось бы рассмотреть подробнее. Начну с обсуждения четырех необязательных составляющих заголовка объекта, показанных на рис. 7.1.

Информация о создателе объекта (OBJECT_CREATOR_INFO)

Если бит OB_FLAG_CREATOR_INFO переменной ObjectFlags установлен, значит, непосредственно перед структурой OBJECT_HEADER в заголовке объекта располагается структура OBJECT_CREATOR_INFO. Определение этой составляющей заголовка объекта приведено в листинге 7.3. Поле ObjectList является элементом двусвязного списка (см. листинг 2.7 в главе 2), соединяющего между собой однотипные объекты. Как и во многих других случаях, список является циклическим. Голова списка, то есть элемент, с которого начинается и которым заканчивается список, входит в состав структуры OBJECT_TYPE. Структура OBJECT_TYPE описывает объект типа, к которому относятся все элементы списка. По умолчанию структуру OBJECT_CREATOR_INFO содержат в составе своего заголовка только объекты Port и WaitablePort. Класс SystemObjectInformation функции ZwQuerySystemInformation() использует ObjectList для того, чтобы вернуть полный список однотипных объектов, размещенных в памяти в текущий момент. В своей книге «Windows NT/2000 Native API Reference» на с. 25 Гари Неббетт (Gary Nebbett) отмечает, что «... этот информационный класс доступен только в случае, если в процессе начальной загрузки был установлен флаг FLG_MAINTAIN_OBJECT_TYPELIST переменной NtGlobalFlags».

Листинг 7.3. Структура OBJECT_CREATOR_INFO

```
typedef struct _OBJECT_CREATOR_INFO
        {
/*000*/    LIST_ENTRY     ObjectList.          //OBJECT_CREATOR_INFO
/*008*/    HANDLE         UniqueProcessId;
/*00C*/    WORD           Reserved1.
/*00E*/    WORD           Reserved2.
/*010*/    }
     OBJECT_CREATOR_INFO.
 *   POBJECT_CREATOR_INFO.
 **  PPOBJECT_CREATOR_INFO.
```

Переменная UniqueProcessId содержит идентификатор процесса, создавшего объект. Нумерация процессов начинается с нуля. Несмотря на то что данный член структуры определен как HANDLE, он не является дескриптором в общепринятом смысле этого слова. Если говорить более точно, это значение является 32-битным числом без знака, точный смысл которого не вполне ясен. На практике входящая в состав стандартного API функция GetCurrentProcessId() возвращает подобные значения типа HANDLE в виде значений типа DWORD.

Имя объекта (OBJECT_NAME)

Если поле NameOffset структуры OBJECT_HEADER отличается от нуля, значит, в нем содержится обратное смещение адреса первой ячейки структуры OBJECT_NAME относительно базового адреса структуры OBJECT_HEADER. Как правило, значение этого смещения составляет 0x10 или 0x20 в зависимости от того, присутствует ли в составе заголовка объекта структура OBJECT_CREATOR_INFO. Определение структуры OBJECT_NAME содержится в листинге 7.4. Поле Name — это структура типа UNICODE_STRING, поле Buffer которой указывает на строку имени, которая, как правило, не является частью блока памяти, в котором располагается объект. Далеко не все именованные объекты хранят собственное имя в разделе OBJECT_NAME собственного заголовка. Многие объекты для получения имени обращаются к функции QueryNameProcedure(), указатель на которую содержится в структуре типа OBJECT_TYPE для данного типа объектов.

Листинг 7.4. Структура OBJECT_NAME

```
typedef struct _OBJECT_NAME
        {
/*000*/    POBJECT_DIRECTORY  Directory;
/*004*/    UNICODE_STRING     Name;
/*00C*/    DWORD              Reserved;
/*010*/    }
      OBJECT_NAME,
 *    POBJECT_NAME,
 **   PPOBJECT_NAME;
```

Если поле Directory структуры OBJECT_NAME не равно NULL, эта переменная указывает на объект каталога, который определяет уровень, на котором данный объект расположен в иерархии системных объектов Windows 2000. Подобно файлам в файловой системе, объекты Windows 2000 организованы в виде иерархического дерева, в состав которого входят *объекты каталога* (directory objects) и *замыкающие объекты* (leaf objects). Более подробное описание структуры OBJECT_DIRECTORY будет приведено далее.

База данных дескрипторов объекта (OBJECT_HANDLE_DB)

В составе некоторых объектов хранится счетчик дескрипторов объекта, выделенных разнообразным работающим в системе процессам. Этот счетчик хранится в так называемой базе данных дескрипторов объекта (handle database). Если объект

обладает такой базой данных, значит, поле HandleDBOffset структуры OBJECT_HEADER не равно нулю и содержит смещение, которое необходимо вычесть из базового адреса структуры OBJECT_HEADER для того, чтобы получить адрес структуры OBJECT_HANDLE_DB, определение которой содержится в листинге 7.5. Если в составе переменной ObjectFlags установлен флаг OB_FLAG_SINGLE_PROCESS, значит, используется член Process объединения, расположенного в начале структуры OBJECT_HANDLE_DB. В этом случае переменная Process указывает на объект процесса. Если дескрипторами объекта обладают несколько процессов, флаг OB_FLAG_SINGLE_PROCESS сброшен и тогда вместо поля Process в составе структуры OBJECT_HANDLE_DB используется поле HandleDBList, которое является указателем на структуру OBJECT_HANDLE_DB_LIST, включающую в себя значение счетчика и массив структур OBJECT_HANDLE_DB.

Листинг 7.5. Структура OBJECT_HANDLE_DB

```
typedef struct _OBJECT_HANDLE_DB
        {
/*000*/    union
             {
/*000*/       struct    _EPROCESS               *Process;
/*000*/       struct    _OBJECT_HANDLE_DB_LIST  *HandleDBList;
/*004*/       };
/*004*/    DWORD HandleCount;
/*00B*/    }
      OBJECT_HANDLE_DB,
  *   POBJECT_HANDLE_DB,
 **   PPOBJECT_HANDLE_DB;

#define OBJECT_HANDLE_DB_ \
 sizeof (OBJECT_HANDLE_DB)

//   ---------------------------------------------------------------

typedef struct _OBJECT_HANDLE_DB_LIST
        {
/*000*/    DWORD              Count;
/*004*/    OBJECT_HANDLE_DB   Entries [];
/*???*/    }
      OBJECT_HANDLE_DB_LIST,
 *    POBJECT_HANDLE_DB_LIST,
 **   PPOBJECT_HANDLE_DB_LIST;
```

Использование ресурсов и квотирование

Когда процесс открывает дескриптор для доступа к объекту, он обязан «заплатить» за использование системных ресурсов, связанных с этой операцией. Плата за использование ресурса обозначается английским термином *charge*. Ограничения, которые операционная система накладывает на использование ресурса, обозначают термином «*квота*» (quota). Другими словами, квота — это максимальный допустимый объем ресурса, который разрешается использовать тому или иному процессу. В документации DDK компания Microsoft определяет термин «квота» следующим образом:

Квота (Quota) — это определенный для конкретного процесса предел на использование тех или иных системных ресурсов. Для каждого процесса Windows NT/Windows 2000 назначает лимит использования тех или иных системных ресурсов. Программные потоки, принадлежащие этому процессу, не могут использовать объем ресурса свыше разрешенного максимально допустимого значения. Квота может быть назначена в отношении файла виртуальной памяти, свопируемой физической памяти, несвопируемой физической памяти и т. п. В частности, диспетчер управления памятью (Memory Manager) взимает с каждого процесса плату каждый раз, когда потоки, принадлежащие этому процессу, обращаются к системе с запросом на выделение виртуальной памяти, физической свопируемой памяти и физической несвопируемой памяти. Диспетчер управления памятью следит за тем, какое количество памяти используется каждым из процессов, не превысило ли это значение установленную для каждого из процессов квоту. По мере того как потоки процесса освобождают используемую ими память, диспетчер памяти обновляет сведения об использовании памяти данным процессом. (Windows 2000 DDK \ Kernel-Mode Drivers \ Design Guide \ Kernel-Mode Glossary \ Q \ quota.)

По умолчанию плата за использование свопируемой и несвопируемой памяти, а также за обращение к системе безопасности определяется в структуре OBJECT_TYPE, соответствующей типу данного объекта. Однако добавив в заголовок объекта раздел OBJECT_QUOTA_CHARGES, вы можете переопределить значения, заданные по умолчанию. Расположение раздела OBJECT_QUOTA_CHARGES относительно структуры OBJECT_HEADER определяется полем QuotaChargesOffset структуры OBJECT_HEADER. Если значение этого поля отличается от нуля, значит, в нем содержится обратное смещение структуры OBJECT_QUOTA_CHARGES относительно начального адреса структуры OBJECT_HEADER. Определение структуры OBJECT_QUOTA_CHARGES содержится в листинге 7.6. За использование свопируемой и несвопируемой памяти плата взимается отдельно. Если объект нуждается в защите, добавляется дополнительная плата за использование системы безопасности, значение которой хранится в поле SecurityCharge. По умолчанию за использование системы безопасности взимается 0x800 условных единиц.

Листинг 7.6. Структура OBJECT_QUOTA_CHARGES

```
#define OB_SECURITY_CHARGE 0x00000800

typedef struct _OBJECT_QUOTA_CHARGES
        {
/*000*/    DWORD PagedPoolCharge;
/*004*/    DWORD NonPagedPoolCharge.
/*008*/    DWORD SecurityCharge;
/*00C*/    DWORD Reserved.
/*010*/    }
      OBJECT_QUOTA_CHARGES.
 *    POBJECT_QUOTA_CHARGES.
 **   PPOBJECT_QUOTA_CHARGES;
```

Если флаг OB_FLAG_CREATE_INFO в составе поля ObjectFlags структуры OBJECT_HEADER равен нулю, поле QuotaBlock указывает на структуру QUOTA_BLOCK (определенную в листинге 7.7), в которой содержится статистическая информация о текущем использовании ресурсов объектом.

Листинг 7.7. Структура QUOTA_BLOCK

```
typedef struct _QUOTA_BLOCK
        {
/*000*/    DWORD     Flags;
/*004*/    DWORD     ChangeCount;
/*008*/    DWORD     PeakPoolUsage  [2]; // несвопируемая память, свопируемая память
/*010*/    DWORD     Flags          [2]. // несвопируемая память, свопируемая память
/*018*/    DWORD     Flags          [2]; // несвопируемая память, свопируемая память
/*020*/    }
     QUOTA_BLOCK.
 *   PQUOTA_BLOCK.
 **  PPQUOTA_BLOCK.
```

Каталог объектов (OBJECT_DIRECTORY)

Как уже отмечалось ранее, при обсуждении структуры OBJECT_NAME, диспетчер объектов Windows 2000 хранит отдельные объекты в составе иерархии структур OBJECT_DIRECTORY. Такие структуры также называют *объектами каталога* (directory objects). Структура OBJECT_DIRECTORY — это еще один любопытный тип объектов, включающий в себя традиционную для всех объектов структуру OBJECT_ HEADER и все остальное, что входит в состав любых других объектов. Управление каталогом объектов Windows 2000 выполняется несколько запутанно. Как видно из листинга 7.8, структура OBJECT_DIRECTORY включает в себя хэш-таблицу, состоящую из 37 записей. Столь необычный размер был выбран, скорее всего, из-за того, что число 37 является простым. Каждая запись таблицы может содержать в себе указатель на структуру OBJECT_DIRECTORY_ENTRY, поле Object которой указывает на объект. Когда создается новый объект, диспетчер объектов на основе имени объекта вычисляет хэш-значение в диапазоне от 0 до 36 и создает структуру OBJECT_DIRECTORY_ ENTRY. Если целевой слот таблицы пуст, в него заносится указатель на новую структуру OBJECT_DIRECTORY_ENTRY. Если слот уже занят, новая запись добавляется в односвязный список записей, начинающийся из целевого слота и связанный при помощи полей NextEntry входящих в него структур. Чтобы сформировать иерархические взаимоотношения между объектами, достаточно разместить в поле Object структуры OBJECT_DIRECTORY_ENTRY указатель на иерархически подчиненную структуру OBJECT_DIRECTORY.

Листинг 7.8. Структуры OBJECT_DIRECTORY и OBJECT_DIRECTORY_ENTRY

```
typedef struct _OBJECT_DIRECTORY_ENTRY
        {
/*000*/    struct _OBJECT_DIRECTORY_ENTRY    *NextEntry.
/*004*/    POBJECT                           Object.
/*008*/    }
     OBJECT_DIRECTORY_ENTRY.
 *   POBJECT_DIRECTORY_ENTRY.
 **  PPOBJECT_DIRECTORY_ENTRY:

//  ------------------------------------------------------------------

#define OBJECT_HASH_TABLE_SIZE 37

typedef struct _OBJECT_DIRECTORY
```

```
        {
/*000*/    POBJECT_DIRECTORY_ENTRY      HashTable  [OBJECT_HASH_TABLE_SIZE].
/*094*/    POBJECT_DIRECTORY_ENTRY      CurrentEntry.
/*098*/    BOOLEAN                      CurrentEntryValid.
/*099*/    BYTE                         Reserved1.
/*09A*/    WORD                         Reserved2;
/*09C*/    DWORD                        Reserved3.
/*0A0*/    }
     OBJECT_DIRECTORY.
 *   POBJECT_DIRECTORY.
 **  PPOBJECT_DIRECTORY.
```

Чтобы оптимизировать доступ к наиболее часто используемым объектам, диспетчер объектов применяет простой алгоритм сортировки объектов по частоте использования, обозначаемый MRU (Most Recently Used). Согласно этому алгоритму каждый раз при успешном обращении к объекту ссылка на соответствующую этому объекту запись OBJECT_DIRECTORY_ENTRY перемещается в начало списка ссылок. Более того, указатель на обновленный список размещается в поле CurrentEntry структуры OBJECT_DIRECTORY. Флаг CurrentEntryValid указывает на то, что указатель CurrentEntry является корректным. Доступ к глобальному каталогу объектов системы синхрониэируется при помощи объекта блокировки типа ERESOURCE с именем ObpRootDirectoryMutex. Этот объект не документирован и не экспортируется.

Типы объектов (OBJECT_TYPE)

Рассказывая о различных составных частях заголовка объекта, я часто упоминал об объектах типов, имея в виду структуры OBJECT_TYPE. Пришло время рассказать о них поподробнее. Объект типа — это специальная разновидность объекта, такого как, например, событие, устройство или процесс. Таким образом, как и все остальные объекты, объект типа включает в себя структуру OBJECT_HEADER, а также, возможно, другие составные части заголовка. Различие состоит в том, что объекты типов связаны с остальными объектами особенным образом. Объект типа — это своего рода образцовый объект, в котором определяются общие свойства объектов этой же категории. Кроме того, в некоторых случаях объект типа содержит в себе двунаправленный список всех объектов данной категории (об этом рассказывалось ранее, при обсуждении структуры OBJECT_CREATOR_INFO). Таким образом, *объекты типов* (type object) часто называют *типами объектов* (object type). Этим подчеркивается то обстоятельство, что объект типов — это нечто большее, чем обычный объект.

Тело объекта типа состоит из структуры OBJECT_TYPE, в состав которой входит структура OBJECT_TYPE_INITIALIZER. Определения обеих этих структур показаны в листинге 7.9. Структура OBJECT_TYPE_INITIALIZER используется для формирования заголовка объекта при создании объекта с использованием функции ObCreateObject(). Например, поля MaintainHandleCount и MaintainTypeList используются внутренней функцией ObpAllocateObject() модуля ntoskrnl.exe для того, чтобы определить, должны ли вновь создаваемые объекты данного типа включать в себя разделы OBJECT_HANDLE и OBJECT_CREATOR_INFO соответственно. Если флаг MaintainTypeList установ-

лен, все объекты данного типа будут автоматически связываться в двунаправленный кольцевой список, началом и окончанием которого является поле ObjectListHead структуры OBJECT_TYPE. Структура OBJECT_TYPE_INITIALIZER содержит также сведения о квотировании и плате за использование ресурсов для данного объекта. Для этого используются поля DefaultPagedPoolCharge и DefaultNonPagedPoolCharge. О квотировании и слежении за использованием системных ресурсов рассказывалось чуть раньше, при обсуждении структуры OBJECT_ QUOTA_CHARGES.

Листинг 7.9. Структуры OBJECT_TYPE и OBJECT_TYPE_INITIALIZER

```
typedef struct _OBJECT_TYPE_INITIALIZER
        {
/*000*/    WORD                  Length;                    //0x004C
/*002*/    BOOLEAN               UseDefaultObject;          //OBJECT_TYPE.DefaultObject
/*003*/    BOOLEAN               Reserved1;
/*004*/    DWORD                 InvalidAttributes;
/*008*/    GENERIC_MAPPING       GenericMapping;
/*018*/    ACCESS_MASK           ValidAccessMask;
/*01C*/    BOOLEAN               SecurityRequired;
/*01D*/    BOOLEAN               MaintainHandleCount;       // OBJECT_HANDLE_DB
/*01E*/    BOOLEAN               MaintainTypeList;          // OBJECT_CREATOR_INFO
/*01F*/    BYTE                  Reserved2;
/*020*/    BOOL                  PagedPool;
/*024*/    DWORD                 DefaultPagedPoolCharge;
/*028*/    DWORD                 DefaultNonPagedPoolCharge;
/*02C*/    NTPROC                DumpProcedure;
/*030*/    NTPROC                OpenProcedure;
/*034*/    NTPROC                CloseProcedure;
/*038*/    NTPROC                DeleteProcedure;
/*03C*/    NTPROC_VOID           ParseProcedure;
/*040*/    NTPROC_VOID           SecurityProcedure;         // SeDefaultObjectMethod
/*044*/    NTPROC_VOID           QueryNameProcedure;
/*048*/    NTPROC_BOOLEAN        OkayToCloseProcedure;
/*04C*/    }
     OBJECT_TYPE_INITIALIZER,
 *   POBJECT_TYPE_INITIALIZER,
 **   PPOBJECT_TYPE_INITIALIZER;

//  ------------------------------------------------------

typedef struct _OBJECT_TYPE
        {
/*000*/    ERESOURCE             Lock;
/*038*/    LIST_ENTRY            ObjectListHead;    // OBJECT_CREATOR_INFO
/*040*/    UNICODE_STRING        ObjectTypeName;    // см. выше
/*04B*/    union
             {
/*048*/        PVOID             DefaultObject;     // ObpDefaultObject
/*048*/        DWORD             Code;              // File: 5C. WaitablePort: A0
             };
/*04C*/    DWORD                     ObjectTypeIndex;       // OB_TYPE_INDEX_*
/*050*/    DWORD                     ObjectCount;
/*054*/    DWORD                     HandleCount;
/*058*/    DWORD                     PeakObjectCount;
/*05C*/    DWORD                     PeakHandleCount;
```

```
/*060*/    OBJECT_TYPE_INITIALIZER ObjectTypeInitializer;
/*0AC*/    DWORD                   ObjectTypeTag;          // OB_TYPE_TAG_*
/*0B0*/    }
     OBJECT_TYPE,
 *   POBJECT_TYPE,
 **  PPOBJECT_TYPE;
```

Объекты типов/типы объектов являются важным строительным материалом вселенной Windows 2000, поэтому модуль ntoskrnl.exe хранит эти объекты в именованных переменных, благодаря чему упрощается процедура определения типа любого из объектов. Если у вас есть объект и вы хотите проверить его тип, достаточно просто сравнить поле ObjectType структуры OBJECT_HEADER интересующего вас объекта с предполагаемым объектом типа. Каждый из объектов типов уникален, то есть для каждого типа объектов существует только один объект типа. Система никогда не создает более чем один объект типа для каждой разновидности объектов. В табл. 7.4 перечислены все объекты типов, используемые в рабочей среде Windows 2000.

Таблица 7.4. Объекты типов

Индекс	Метка	Имя	Структура C	Публичный	Символ
1	"ObjT"	"Type"	OBJECT_TYPE	Нет	ObpTypeObjectType
2	"Dire"	"Directory"	OBJECT_DIRECTORY	Нет	ObpDirectoryObjectType
3	"Symb"	"SymbolicLink"		Нет	ObpSymbolicLinkObject-Type
4	"Toke"	"Token"	TOKEN	Нет	SepTokenObjectType
5	"Proc"	"Process"	EPROCESS	Да	PsProcessType
6	"Thre"	"Thread"	ETHREAD	Да	PsThreadType
7	"Job "	"Job"		Да	PsJobType
8	"Even"	"Event"	KEVENT	Да	ExEventObjectType
9	"Even"	"EventPair"	KEVENT_PAIR	Нет	ExEventPairObjectType
10	"Muta"	"Mutant"	KMUTANT	Нет	ExMutantObjectType
11	"Call"	"CallBack"	CALLBACK_OBJECT	Нет	ExCallbackObjectType
12	"Sema"	"Semaphore"	KSEMAPHORE	Да	ExSemaphoreObjectType
13	"Time"	"Timer"	ETIMER	Нет	ExTimerObjectType
14	"Prof"	"Profile"	KPROFILE	Нет	ExProfileObjectType
15	"Wind"	"WindowStation"		Да	ExWindowStation-ObjectType
16	"Desk"	"Desktop"		Да	ExDesktopObjectType
17	"Sect"	"Section"		Да	MmSectionObjectType
18	"Key "	"Key"		Нет	CmpKeyObjectType
19	"Port"	"Port"		Да	LpcPortObjectType
20	"Wait"	"WaitablePort"		Нет	LpcWaitablePortObjectType
21	"Adap"	"Adapter"	ADAPTER_OBJECT	Да	IoAdapterObjectType
22	"Cont"	"Controller"	CONTROLLER_OBJECT	Нет	IoControllerObjectType
23	"Devi"	"Device"	DEVICE_OBJECT	Да	IoDeviceObjectType
24	"Driv"	"Driver"	DRIVER_OBJECT	Да	IoDriverObjectType

продолжение

Таблица 7.4 (*продолжение*)

Индекс	Метка	Имя	Структура С	Публичный	Символ
25	"IoCo"	"IoCompletion"	IO_COMPLETION	Нет	IoCompletionObjectType
26	"File"	"File"	FILE_OBJECT	Да	IoFileObjectType
27	"WmiG"	"WmiGuid"	GUID	Нет	WmipGuidObjectType

Колонки табл. 7.4 имеют следующий смысл.

- **Индекс** — это значение поля ObjectTypeIndex структуры OBJECT_TYPE.
- **Метка** — 32-битный идентификатор, размещаемый в поле ObjectTypeTag структуры OBJECT_TYPE. В Windows 2000 метками, как правило, являются бинарные значения, представляющие собой комбинацию из четырех символов ANSI. В процессе отладки эти символы можно легко обнаружить, просматривая содержимое памяти в виде шестнадцатеричного листинга. При помощи метки чрезвычайно удобно идентифицировать тип объекта. Чтобы проверить, принадлежит ли некоторый объект типа к тому или иному типу, достаточно выполнить одну операцию сравнения поля ObjectTypeTag и 32-битного значения метки предполагаемого типа. Выделяя память для объекта, операционная система Windows 2000 также использует значение метки — она логически складывает его с константой 0x80000000 и метит полученным числом новый блок памяти.
- **Имя** — это имя объекта, указываемое в составе раздела OBJECT_NAME заголовка объекта. Как можно заметить, метка типа формируется на основе имени типа путем отсечения первых четырех символов и, в случае необходимости (если имя типа состоит из менее чем четырех символов), добавления к ним пробелов.
- **Структура С** — имя структуры, представляющей собой тело объекта данного типа. Некоторые из упомянутых в таблице структур документированы в DDK, а некоторые определены в заголовочном файле w2k_def.h, записанном на прилагаемом к данной книге компакт-диске. Если в строке таблицы отсутствует какое-либо имя структуры, значит, на момент написания данной книги структура неизвестна или не идентифицирована.
- **Символ** — это имя переменной-указателя, которая ссылается на соответствующий объект типа. Если в колонке «Публичный» содержится слово «Да», значит, переменная экспортируется и к ней можно получить доступ из драйверов режима ядра или из приложений, работающих с ядром при помощи библиотеки w2k_call.dll, описанной в главе 6.

Колонка «Индекс» требует дополнительных пояснений. Показанное здесь значение изъято из поля ObjectTypeIndex соответствующей структуры OBJECT_TYPE. Это значение *не* является заранее определенным идентификатором типа (Type ID), таким как константы DISP_TYPE_* и IO_TYPE_*, используемые объектами синхронизации и объектами ввода/вывода (см. табл. 7.1 и 7.2). Индекс, хранящийся в поле ObjectTypeIndex, всего лишь отражает последовательность, в которой система создает перечисленные в таблице объекты типов. Исходя из этого не следует использовать поле ObjectTypeIndex для идентификации типа объекта. Вместо этого для идентификации типов безопаснее использовать метку ObjectTypeTag, так как, вероятнее всего, в будущих версиях Windows компания Microsoft не станет менять существующие метки типов.

Дескрипторы объектов

В то время как драйвер режима ядра может напрямую взаимодействовать с объектом, запросив у системы указатель на тело этого объекта, приложение пользовательского режима не обладает такой возможностью. Вместо этого оно обращается к одной из функций API, которая служит для открытия объекта, и получает от этой функции *дескриптор* (handle) объекта. В дальнейшем, когда приложение выполняет какие-либо операции, связанные с открытым объектом, оно идентифицирует объект при помощи этого дескриптора. В терминологии Windows 2000 термином *дескриптор* (handle) обозначается множество разных, подчас никак не связанных между собой вещей, однако существует конструкция, которую можно считать дескриптором в самом строгом смысле этого слова. Дескриптор, о котором идет речь, является выделяемым отдельно для каждого процесса 16-битным числом, которое обычно кратно четырем и представляет собой индекс в таблице дескрипторов, которую ядро формирует для каждого работающего в системе процесса. Определение главной структуры HANDLE_TABLE показано в конце листинга 7.10. Эта таблица указывает на структуру HANDLE_LAYER1, которая состоит из указателей на структуры HANDLE_LAYER2, каждая из которых состоит из указателей HANDLE_LAYER3. Наконец, на третьем уровне перенаправления располагаются указатели на записи таблицы дескрипторов, каждая из которых представлена структурой HANDLE_ENTRY.

Листинг 7.10. Таблицы дескрипторов, уровни и записи

```
// HANDLE BIT-FIELDS (БИТОВЫЕ ПОЛЯ ДЕСКРИПТОРА)
//  ---------------------------------------
//  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
//  F E D C B A 9 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 6 5 4 3 2 1 0
//
//  |x|x|x|x|x|x|a|a|a|a|a|a|a|a|b|b|b|b|b|b|b|b|c|c|c|c|c|c|c|c|y|y|
// | not used  | HANDLE_LAYER1 | HANDLE_LAYER2 | HANDLE_LAYER3 |tag|

#define HANDLE_LAYER_SIZE 0x00000100

//  ------------------------------------------------------------------

#define HANDLE_ATTRIBUTE_INHERIT   0x00000002
#define HANDLE_ATTRIBUTE_MASK      0x00000007
#define HANDLE_OBJECT_MASK         0xFFFFFFF8

typedef struct _HANDLE_ENTRY  // см. OBJECT_HANDLE_INFORMATION
        {
/*000*/   union
            {
/*000*/       DWORD           HandleAttributes; // HANDLE_ATTRIBUTE_MASK
/*000*/       POBJECT_HEADER  ObjectHeader;     // HANDLE_OBJECT_MASK
/*004*/       };
/*004*/   union
            {
/*004*/       ACCESS_MASK     GrantedAccess;    // Если запись используется
/*004*/       DWORD           NextEntry;        // Если запись пуста
/*008*/       };
/*008*/   }
    HANDLE_ENTRY,
 *  PHANDLE_ENTRY,
 ** PPHANDLE_ENTRY;
```

продолжение

Листинг 7.10. Таблицы дескрипторов, уровни и записи

```
//  ------------------------------------------------------------------

typedef struct _HANDLE_LAYER3
        {
/*000*/    HANDLE_ENTRY  Entries [HANDLE_LAYER_SIZE]. // биты от 2 до 9
/*800*/    }
     HANDLE_LAYER3.
 *   PHANDLE_LAYER3.
 **  PPHANDLE_LAYER3;

//  ------------------------------------------------------------------

typedef struct _HANDLE_LAYER2
        {
/*000*/    PHANDLE_LAYER3 Layer3 [HANDLE_LAYER_SIZE]; // биты от 10 до 17
/*400*/    }
     HANDLE_LAYER2.
 *   PHANDLE_LAYER2.
 **  PPHANDLE_LAYER2;

//  ------------------------------------------------------------------

typedef struct _HANDLE_LAYER1
        {
/*000*/    PHANDLE_LAYER2 Layer2 [HANDLE_LAYER_SIZE]; // биты от 1B до 25
/*400*/    }
     HANDLE_LAYER1.
 *   PHANDLE_LAYER1.
 **  PPHANDLE_LAYER1;

//  ------------------------------------------------------------------

typedef struct _HANDLE_TABLE
        {
/*000*/    DWORD             Reserved;
/*004*/    DWORD             HandleCount;
/*008*/    PHANDLE_LAYER1    Layer1
/*00C*/    struct _EPROCESS  *Process,          // передается в PsChargePoolQuota ()
/*010*/    HANDLE            UniqueProcessId;
/*014*/    DWORD             NextEntry;
/*018*/    DWORD             TotalEntries.
/*01C*/    ERESOURCE         HandleTableLock;
/*054*/    LIST_ENTRY        HandleTableList;
/*05C*/    KEVENT            Event.
/*06C*/    }
     HANOLE_TABLE.
 *   PHANDLE_TABLE.
 **  PPHANDLE_TABLE.
```

Зачем потребовалось формировать таблицу дескрипторов в виде столь сложной трехуровневой конструкции? На самом деле описанный трехуровневый механизм адресации является хитрым способом динамически увеличивать и уменьшать пространство в памяти, необходимое для хранения записей о дескрипторах. При этом для изменения размеров таблицы требуются минимальные усилия, а непро-

дуктивный расход памяти минимизируется. На каждом уровне таблицы дескрипторов может храниться до 256 указателей, это значит, что один процесс теоретически может открыть 256*256*256 или 16 777 216 дескрипторов. Каждая запись о дескрипторе занимает 8 байт, а это значит, что для хранения всех этих дескрипторов потребуется 128 Мбайт. Однако подавляющему большинству процессов не требуется использовать такое огромное количество дескрипторов, поэтому выделение памяти для хранения абсолютно всех этих дескрипторов было бы непродуктивным. При использовании трехуровневого подхода система выделяет минимально допустимое количество памяти на каждом уровне. Если не считать самой структуры HANDLE_TABLE, для хранения такой минимальной таблицы дескрипторов требуется 256*4+256*4+256*8 или 4096 байт. Таким образом, изначально таблица дескрипторов умещается в одну физическую страницу памяти.

Чтобы получить структуру HANDLE_ENTRY, соответствующую значению HANDLE, система делит дескриптор на три 8-битных фрагмента, при этом биты 0 и 1, а также старшие шесть бит дескриптора не рассматриваются. Получив в распоряжение эти три составляющие дескриптора, механизм сопоставления записей о дескрипторах действует следующим образом:

1. Биты от 18 до 25 значения HANDLE используются в качестве индекса при обращении к массиву Layer2 блока HANDLE_LAYER1, на который ссылается поле Layer1 структуры HANDLE_TABLE.
2. Биты от 10 до 17 значения HANDLE используются в качестве индекса при обращении к массиву Layer3 блока HANDLE_LAYER2, адрес которого получен на предыдущем шаге.
3. Биты от 2 до 9 значения HANDLE используются в качестве индекса при обращении к массиву Entries блока HANDLE_LAYER3, адрес которого получен на предыдущем шаге.
4. Полученная на предыдущем шаге структура HANDLE_ENTRY содержит указатель на структуру OBJECT_HEADER (см. листинг 7.2) объекта, связанного с дескриптором HANDLE.

Если описанная схема кажется вам запутанной, обратите внимание на рис. 7.2, изучив который можно образно представить себе работу механизма сопоставления записей дескрипторов. Можно заметить, что схема на рис. 7.2 сильно напоминает рис. 4.3 в главе 4, в которой речь шла о механизме трансляции адресов, который используется в семействе процессоров i386 для преобразования линейного адреса в физический. Оба алгоритма разбивают исходное значение на три составные части, две из которых используются в качестве смещений при выборе записей на двух иерархически связанных уровнях перенаправления, а третья составная часть используется для выбора записи на целевом уровне. Многоуровневая модель таблицы дескрипторов впервые появилась в Windows 2000. В операционной системе Windows NT 4.0 использовалась одноуровневая таблица, которую необходимо было расширять в случае, если вновь открытый дескриптор не умещается в блок памяти, выделенный для хранения таблицы (подробнее об этом написано в книгах Custer H. «Inside the Windows NT». Redmond, WA: Microsoft Press, 1993 и Solomon D. A. «Inside Windows NT». 2nd ed. Redmond, WA: Microsoft Press, 1998).

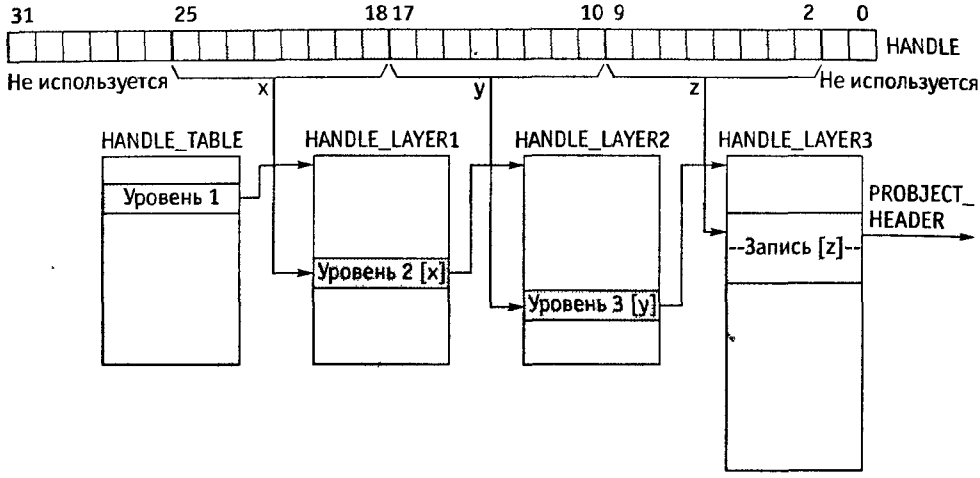

Рис. 7.2. Получение указателя на заголовок объекта OBJECT_HEADER исходя из дескриптора HANDLE

Каждый процесс обладает собственной таблицей дескрипторов, поэтому ядро должно обладать возможностью следить за выделенными на текущий момент таблицами. Для этой цели в модуле ntoskrnl.exe используется переменная HandleTableListHead типа LIST_ENTRY, которая является корнем двусвязного списка структур HANDLE_TABLE, связанных между собой при помощи полей HandleTableList. Следуя по указателю Flink или Blink структуры типа LIST_ENTRY, для того чтобы получить базовый адрес соответствующей структуры HANDLE_TABLE, необходимо всегда вычитать из полученного значения смещение поля HandleTableList, равное 0x54. Чтобы определить процесс, которому принадлежит некоторая таблица, необходимо проанализировать поле UniqueProcessId соответствующей структуры HANDLE_ TABLE. Первая таблица в списке, как правило, принадлежит процессу System (ID=8). За ним в большинстве случаев следует таблица процесса холостого хода System Idle Process (ID=0). Доступ к этой таблице можно получить также при помощи внутренней переменной ядра ObpKernelHandleTable.

Чтобы обеспечить целостность данных при доступе к дескрипторам в многопоточной операционной среде, система использует пару объектов синхронизации. Список таблиц дескрипторов блокируется при помощи глобального объекта HandleTableListLock, расположенного внутри модуля ntoskrnl.exe и являющегося структурой типа ERESOURCE. Этот тип объекта синхронизации поддерживает как эксклюзивное (exclusive), так и общее (shared) блокирование, которое выполняется с использованием функций ExAcquireResourceExclusiveLite() и ExAcquireResourceSharedLite() соответственно. Снять блокировку можно при помощи функции ExReleaseResourceLite(). Если вы блокировали список таблиц дескрипторов для эксклюзивного доступа, вы можете быть уверенными в том, что система не изменит ни одну из записей списка до тех пор, пока вы не разблокируете список. Каждая структура HANDLE_TABLE обладает собственным объектом блокировки, который в листинге 7.10 обозначен как HandleTableLock. Модуль ntoskrnl.exe обладает собственными внутренними функциями ExLockHandleTableExclusive() и ExLockHandleTableShared(), при помощи кото-

рых можно получить в собственное пользование объект ERESOURCE блокирования таблицы дескрипторов. Для разблокирования таблицы в ядре применяется функция ExUnlockHandleTableShared(). Несмотря на то что в имени функции присутствует слово Shared, эта функция используется для разблокирования таблицы вне зависимости от того, была ли таблица дескрипторов блокирована для эксклюзивного или для общего доступа. Все эти функции ядра для выполнения блокирования и разблокирования таблицы просто обращаются к функциям ExAcquireResourceExclusiveLite(), ExAcquireResourceSharedLite() и ExReleaseResourceLite(). При этом они принимают указатель на HANDLE_TABLE и передают HandleTableLock.

К сожалению, все наиболее важные функции и глобальные переменные, используемые диспетчером дескрипторов, не только не документированы, но и недоступны, так как модуль ntoskrnl.exe не экспортирует их. Конечно же вы можете разыскивать объекты при помощи их дескрипторов, используя интерфейс обращения к ядру, описанный в главе 6, и алгоритм, проиллюстрированный на рис. 7.2, однако я не рекомендую так поступать. Во-первых, разработанный вами код не будет совместим с Windows NT 4, так как в этой системе таблица дескрипторов устроена совершенно по-другому. Во-вторых, вместо того чтобы корпеть над разработкой собственного механизма доступа к таблице дескрипторов, можно воспользоваться весьма удобной функцией ядра, специально предназначенной для получения содержимого всех таблиц дескрипторов, принадлежащих текущему активному процессу. Эта функция имеет имя NtQuerySystemInformation(), она размещает информацию о дескрипторе в классе с именем SystemHandleInformation. Подробная дискуссия о том, как можно использовать данную функцию, содержится в книге Неббета (Nebbet G. «Windows NT/2000 Native API Reference». Indianapolis, IN: Macmillan Technical Publishing (MTP), 2000) и в моей статье (Schreiber S. B. «Inside Windows NT System Data». «Dr. Dobb's Journal», no. 305, November (1999). San Francisco, CA: CMP Media исходный код можно найти по адресу http://www.ddj.com/ftp/1999/1999_11/ntinfo.zip). Данные для структуры SystemHandleInformation извлекаются при помощи внутренней функции ExpGetHandleInformation(), которая использует функцию ObGetHandleInformation(). А эта функция, в свою очередь, обращается к вызову ExSnapShotHandleTables(), в рамках которой, собственно, и происходит последовательный перебор списка таблиц дескрипторов. В качестве одного из аргументов функция ExSnapShotHandleTable() принимает указатель на функцию, которая вызывается для каждой записи HANDLE_ENTRY, указывающей на объект. В качестве функции обработки записи HANDLE_ENTRY функция ObGetHandleInformation() использует внутренний вызов ObpCaptureHandleInformation(), который заполняет предоставленный вызвавшим процессом буфер массивом структур, содержащих информацию о каждом из дескрипторов, с которыми в настоящий момент работает система.

Объекты процессов и потоков

Возможно, наиболее интересными обитателями мира объектов Windows 2000 являются объекты процессов и потоков. Эти объекты обладают наиболее сложной структурой, и именно с этими объектами должен иметь дело разработчик программного обеспечения. Компонент режима ядра всегда выполняется в контексте программного потока, а этот поток часто является составной частью пользова-

тельского процесса. Исходя из этого именно объекты потоков и процессов чаще всего приходится исследовать в процессе отладки программного обеспечения. Отладчик ядра Windows 2000 Kernel Debugger обладает двумя командами, специально предназначенными для этой цели. Команды !processfields и !threadfields экспортируются расширением kdextx86.dll отладчика. Обе команды выводят на экран простой список пар имя/смещение, описывающих поля структур EPROCESS и ETHREAD соответственно (см. примеры 1.1 и 1.2 в главе 1). Строение этих структур не документировано, поэтому упомянутые команды отладчика являются фактически единственным официальным источником информации о них.

К великому сожалению, команда !processfields не дает никакой информации о строении самого первого поля структуры EPROCESS с именем Pcb. Поле с этим именем располагается в самом начале списка и состоит из 0x6C байт (см. пример 1.1). Pcb — это структура KPROCESS, строение которой полностью не документировано. Со всей очевидностью можно сказать, что процесс представлен в виде большого объекта EPROCESS модуля Executive, в состав которого входит маленький объект KPROCESS, с которым работает ядро. Аналогичная схема вложения объектов используется при формировании объекта программного потока. Изучив вывод команды !threadfields отладчика (см. пример 1.2), можно обнаружить в самом начале списка полей структуры ETHREAD переменную с именем Tcb, включающую в себя 0x1B0 байт. Tcb — это структура KTHREAD, которая является еще одним объектом уровня ядра, входящим в состав объекта уровня Executive.

Конечно же, информация об объектах процесса и потока, отображаемая отладчиком, весьма любопытна, однако даже зная имена и смещения структур, сложно судить о типе данных и предназначении того или иного поля. Мало того, совершенная закрытость полей Pcb и Tcb не позволяет сделать ни единой догадки о внутренней природе этих объектов. Изучая дизассемблерный листинг, выдаваемый отладчиком ядра, вы можете обратить внимание на инструкции, обращающиеся к данным, расположенным в границах этих закрытых членов. Используемые при этом смещения фактически полностью бесполезны, так как если вы не знаете имен и типов данных, с которыми выполняются операции, вы не можете делать никаких полезных выводов. Чтобы понять, каким образом выглядят эти объекты, я собрал разрозненные сведения из самых разных источников, а также провел собственные исследования. Частичный результат моих изысканий представлен в листингах 7.11 и 7.12, в которых содержатся определения структур KPROCESS и KTHREAD соответственно. Структура DISPATCHER_HEADER в начале обоих объектов свидетельствует о том, что оба этих объекта являются объектами синхронизации. А это, в свою очередь, значит, что изменения их состояния можно ожидать при помощи функций KeWaitForSingleObject() и KeWaitForMultipleObjects(). Объект потока переходит в сигнальное состояние после того, как выполнение потока прекращается, а процесс переходит в сигнальное состояние после того, как все его потоки завершают работу. Это обстоятельство не является новым для подавляющего большинства программистов Win32 — ожидание при помощи стандартной функции WaitForSingleObject() завершения работы процесса, запущенного из другого процесса, является широко распространенным приемом. Однако теперь вы, должно быть, понимаете, благодаря чему в рабочей среде Windows 2000 возможно ожидание процессов и потоков.

Листинг 7.11. Структура KPROCESS

```
typedef struct _KPROCESS
        {
/*000*/    DISPATCHER_HEADER       Header,           // DO_TYPE_PROCESS (0x1B)
/*010*/    LIST_ENTRY              ProfileListHead;
/*018*/    DWORD                   DirectoryTableBase;
/*01C*/    DWORD                   PageTableBase;
/*020*/    KGDTENTRY               LdtDescriptor;
/*028*/    KIDTENTRY               Int21Descriptor;
/*030*/    WORD                    IopmOffset;
/*032*/    BYTE                    Iopl,
/*033*/    BOOLEAN                 VdmFlag;
/*034*/    DWORD                   ActiveProcessors,
/*038*/    DWORD                   KernelTime;       // в тиках (ticks)
/*03C*/    DWORD                   UserTime,         // в тиках (ticks)
/*040*/    LIST_ENTRY              ReadyListHead;
/*048*/    LIST_ENTRY              SwapListEntry;
/*050*/    LIST_ENTRY              ThreadListHead;   // KTHREAD.ThreadListEntry
/*058*/    PVOID                   ProcessLock,
/*05C*/    KAFFINITY               Affinity,
/*060*/    WORD                    StackCount,
/*062*/    BYTE                    BasePriority;
/*063*/    BYTE                    ThreadQuantum;
/*064*/    BOOLEAN                 AutoAlignment;
/*065*/    BYTE                    State;
/*066*/    BYTE                    ThreadSeed,
/*067*/    BOOLEAN                 DisableBoost,
/*068*/    DWORD                   d068,
/*06C*/    }
     KPROCESS,
 *   PKPROCESS,
 **  PPKPROCESS;
```

Листинг 7.12. Структура KTHREAD

```
typedef struct _KTHREAD
        {
/*000*/    DISPATCHER_HEADER       Header;           // DO_TYPE_THREAD (0x6C)
/*010*/    LIST_ENTRY              MutantListHead;
/*018*/    PVOID                   InitialStack,
/*01C*/    PVOID                   StackLimit;
/*020*/    struct _TEB             *Teb,
/*024*/    PVOID                   TlsArray;
/*028*/    PVOID                   KernelStack;
/*02C*/    BOOLEAN                 DebugActive;
/*02D*/    BYTE                    State;            // THREAD_STATE_*
/*02E*/    BOOLEAN                 Alerted;
/*02F*/    BYTE                    bReserved01;
/*030*/    BYTE                    Iopl;
/*031*/    BYTE                    NpxState;
/*032*/    BYTE                    Saturation,
/*033*/    BYTE                    Priority,
/*034*/    KAPC_STATE              ApcState,
/*04C*/    DWORD                   ContextSwitches;
/*050*/    DWORD                   WaitStatus;
/*054*/    BYTE                    WaitIrql,
/*055*/    BYTE                    WaitMode,
```

продолжение

Листинг 7.12 (*продолжение*)

```
/*056*/    BYTE                    WaitNext;
/*057*/    BYTE                    WaitReason;
/*058*/    PLIST_ENTRY             WaitBlockList;
/*05C*/    LIST_ENTRY              WaitListEntry;
/*064*/    DWORD                   WaitTime;
/*068*/    BYTE                    BasePriority;
/*069*/    BYTE                    DecrementCount;
/*06A*/    BYTE                    PriorityDecrement;
/*06B*/    BYTE                    Quantum;
/*06C*/    KWAIT_BLOCK             WaitBlock [4];
/*0CC*/    DWORD                   LegoData;
/*0D0*/    DWORD                   KernelApcDisable;
/*0D4*/    KAFFINITY               UserAffinity;
/*0D8*/    BOOLEAN                 SystemAffinityActive;
/*0D9*/    BYTE                    Pad [3].
/*0DC*/    PSERVICE_DESCRIPTOR_TABLE pServiceDescriptorTable;
/*0E0*/    PVOID                   Queue;
/*0E4*/    PVOID                   ApcQueueLock;
/*0E8*/    KTIMER                  Timer;
/*110*/    LIST_ENTRY              QueueListEntry;
/*118*/    KAFFINITY               Affinity;
/*11C*/    BOOLEAN                 Preempted;
/*11D*/    BOOLEAN                 ProcessReadyQueue;
/*11E*/    BOOLEAN                 KernelStackResident;
/*11F*/    BYTE                    NextProcessor.
/*120*/    PVOID                   CallbackStack.
/*124*/    struct _WIN32_THREAD    *Win32Thread;
/*128*/    PVOID                   TrapFrame;
/*12C*/    PKAPC_STATE             ApcStatePointer;
/*130*/    PVOID                   p130
/*134*/    BOOLEAN                 EnableStackSwap;
/*135*/    BOOLEAN                 LargeStack;
/*136*/    BYTE                    ResourceIndex;
/*137*/    KPROCESSOR_MODE         PreviousMode;
/*138*/    DWORD                   KernelTime;          // в тиках (ticks)
/*13C*/    DWORD                   UserTime;            // в тиках (ticks)
/*140*/    KAPC_STATE              SavedApcState;
/*157*/    BYTE                    bReserved02;
/*158*/    BOOLEAN                 Alertable;
/*159*/    BYTE                    ApcStateIndex;
/*15A*/    BOOLEAN                 ApcQueueable;
/*15B*/    BOOLEAN                 AutoAlignment;
/*15C*/    PVOID                   StackBase;
/*160*/    KAPC                    SuspendApc;
/*190*/    KSEMAPHORE              SuspendSemaphore;
/*1A4*/    LIST_ENTRY              ThreadListEntry;     // см. KPROCESS
/*1AC*/    BYTE                    FreezeCount;
/*1AD*/    BYTE                    SuspendCount;
/*1AE*/    BYTE                    IdealProcessor;
/*1AF*/    BOOLEAN                 DisableBoost;
/*1B0*/    }
     KTHREAD.
 *   PKTHREAD.
 **   PPKTHREAD;
```

Объект KPROCESS связан с принадлежащими ему потоками при помощи поля ThreadListHead, которое является начальной и конечной точкой двусвязного списка объектов KTHREAD. Элементами списка являются поля ThreadListEntry. Как и обычно, если речь идет о переменных типа LIST_ENTRY, чтобы получить базовый адрес объекта, в состав которого входит данная переменная, необходимо вычесть из адреса этой переменной значение смещения этой переменной в составе структуры объекта. Это связано с тем, что поля Flink и Blink переменной типа LIST_ ENTRY всегда указывают на следующую переменную типа LIST_ENTRY, входящую в состав списка, а не на базовый адрес структуры, которая включает в себя переменную LIST_ENTRY в качестве одного из полей. Благодаря этому можно перекрестно связывать объекты в составе нескольких разных списков.

В листингах 7.11 и 7.12, равно как и в последующих листингах, можно встретить поля с именами, состоящими из маленькой латинской буквы и следующим за ней трехзначным шестнадцатеричным числом. Подобным образом я обозначаю поля, назначение и тип которых мне неизвестны. Стоящая в начале имени буква обозначает предполагаемый тип поля (например, буква d обозначает DWORD, а буква p — PVOID), а следующее за буквой число является смещением поля относительно начала структуры.

Строение объектов EPROCESS и ETHREAD, являющихся объектами модуля Executive, и содержащими в своем составе объекты ядра KPROCESS и KTHREAD, показано в листингах 7.13 и 7.14. В составе этих структур содержится множество не идентифицированных членов, предназначение которых, я надеюсь, в ближайшем будущем будет установлено. Возможно, исследованием этих полей займутся некоторые читатели этой книги. Однако наиболее важные и наиболее часто используемые члены структур идентифицированы, кроме того, известно, какая информация упущена.

Листинг 7.13. Структура EPROCESS

```
typedef struct _EPROCESS
        {
/*000*/    KPROCESS            Pcb;
/*06C*/    NTSTATUS            ExitStatus;
/*070*/    KEVENT              LockEvent;
/*080*/    DWORD               LockCount;
/*084*/    DWORD               d084;
/*088*/    LARGE_INTEGER       CreateTime;
/*090*/    LARGE_INTEGER       ExitTime;
/*098*/    PVOID               LockOwner;
/*09C*/    DWORD               UniqueProcessId;
/*0A0*/    LIST_ENTRY          ActiveProcessLinks;
/*0A8*/    DWORD               QuotaPeakPoolUsage [2];  // NP (не свопируемая память).
                                                         // P (свопируемая память)
/*0B0*/    DWORD               QuotaPoolUsage     [2];  // NP (не свопируемая память).
                                                         // P (свопируемая память)
/*0B8*/    DWORD               PagefileUsage;
/*0BC*/    DWORD               CommitCharge;
/*0C0*/    DWORD               PeakPagefileUsage;
/*0C4*/    DWORD               PeakVirtualSize;
/*0CB*/    LARGE_INTEGER       VirtualSize;
/*0D0*/    MMSUPPORT           Vm;
```

продолжение

Листинг 7.13 (*продолжение*)

```
/*100*/    DWORD              d100;
/*104*/    DWORD              d104;
/*108*/    DWORD              d108;
/*10C*/    DWORD              d10C;
/*110*/    DWORD              d110;
/*114*/    DWORD              d114;
/*118*/    DWORD              d118;
/*11C*/    DWORD              d11C;
/*120*/    PVOID              DebugPort;
/*124*/    PVOID              ExceptionPort;
/*128*/    PHANDLE_TABLE      ObjectTable;
/*12C*/    PVOID              Token;
/*130*/    FAST_MUTEX         WorkingSetLock;
/*150*/    DWORD              WorkingSetPage;
/*154*/    BOOLEAN            ProcessOutswapEnabled;
/*155*/    BOOLEAN            ProcessOutswapped;
/*156*/    BOOLEAN            AddressSpaceInitialized;
/*157*/    BDOLEAN            AddressSpaceDeleted;
/*158*/    FAST_MUTEX         AddressCreationLock;
/*178*/    KSPIN_LOCK         HyperSpaceLock;
/*17C*/    DWORD              ForkInProgress;
/*180*/    WORD               VmOperation;
/*182*/    BOOLEAN            ForkWasSuccessful;
/*183*/    BYTE               MmAgressiveWsTrimMask;
/*184*/    DWORD              VmOperationEvent;
/*188*/    HARDWARE_PTE       PageDirectoryPte;
/*18C*/    DWORD              LastFaultCount;
/*190*/    DWORD              ModifiedPageCount;
/*194*/    PVOID              VadRoot;
/*198*/    PVOID              VadHint;
/*19C*/    PVOID              CloneRoot;
/*1A0*/    DWORD              NumberOfPrivatePages;
/*1A4*/    DWORD              NumberOfLockedPages;
/*1A8*/    WORD               NextPageColor;
/*1AA*/    BOOLEAN            ExitProcessCalled;
/*1AB*/    BOOLEAN            CreateProcessReported;
/*1AC*/    HANDLE             SectionHandle;
/*1B0*/    struct _PEB        *Peb;
/*1B4*/    PVOID              SectionBaseAddress;
/*1B8*/    PQUOTA_BLOCK       QuotaBlock;
/*1BC*/    NTSTATUS           LastThreadExitStatus;
/*1C0*/    DWORD              WorkingSetWatch;
/*1C4*/    HANDLE             Win32WindowStation;
/*1C8*/    DWORD              InheritedFromUniqueProcessId;
/*1CC*/    ACCESS_MASK        GrantedAccess;
/*1D0*/    DWORD              DefaultHardErrorProcessing;      // HEM_*
/*1D4*/    DWORD              LdtInformation;
/*1DB*/    PVOID              VadFreeHint;
/*1DC*/    DWORD              VdmObjects;
/*1E0*/    PVOID              DeviceMap;                       // 0x24 байт
/*1E4*/    DWORD              SessionId;
/*1E8*/    DWORD              d1E8;
/*1EC*/    DWORD              d1EC;
/*1F0*/    DWORD              d1F0;
/*1F4*/    DWORD              d1F4;
```

```
/*1F8*/    DWORD               d1F8;
/*1FC*/    BYTE                ImageFileName [16];
/*20C*/    DWORD               VmTrimFaultValue;
/*210*/    BYTE                SetTimerResolution;
/*211*/    BYTE                PriorityClass;
/*212*/    union
           {
           struct
               {
/*212*/                              BYTE SubSystemMinorVersion;
/*213*/                              BYTE SubSystemMajorVersion;
               };
           struct
               {
/*212*/                              WORD SubSystemVersion;
               };
           };
/*214*/    struct _WIN32_PROCESS   *Win32Process;
/*218*/    DWORD                   d218;
/*21C*/    DWORD                   d21C;
/*220*/    DWORD                   d220;
/*224*/    DWORD                   d224;
/*228*/    DWORD                   d228;
/*22C*/    DWORD                   d22C;
/*230*/    PVOID                   Wow64;
/*234*/    DWORD                   d234;
/*238*/    IO_COUNTERS             IoCounters;
/*268*/    DWORD                   d268;
/*26C*/    DWORD                   d26C;
/*270*/    DWORD                   d270;
/*274*/    DWORD                   d274;
/*278*/    DWORD                   d278;
/*27C*/    DWORD                   d27C;
/*280*/    DWORD                   d280;
/*284*/    DWORD                   d284;
/*288*/    }
      EPROCESS.
 *    PEPROCESS.
 **   PPEPROCESS;
```

Листинг 7.14. Структура ETHREAD

```
typedef struct _ETHREAD
           {
/*000*/    KTHREAD             Tcb;
/*1B0*/    LARGE_INTEGER       CreateTime;
/*1B8*/    union
           {
/*1B8*/    LARGE_INTEGER       ExitTime;
/*1B8*/    LIST_ENTRY          LpcReplyChain;
           };
/*1C0*/    union
           {
/*1C0*/    NTSTATUS            ExitStatus;
/*1C0*/    DWORD               OfsChain;
           };
/*1C4*/    LIST_ENTRY          PostBlockList;
```

продолжение

Листинг 7.14 (*продолжение*)

```
/*1CC*/    LIST_ENTRY          TerminationPortList.
/*1D4*/    PVOID               ActiveTimerListLock.
/*1D8*/    LIST_ENTRY          ActiveTimerListHead.
/*1E0*/    CLIENT_ID           Cid
/*1E8*/    KSEMAPHORE          LpcReplySemaphore.
/*1FC*/    DWORD               LpcReplyMessage.
/*200*/    DWORD               LpcReplyMessageId;
/*204*/    DWORD               PerformanceCountLow;
/*208*/    DWORD               ImpersonationInfo;
/*20C*/    LIST_ENTRY          IrpList.
/*214*/    PVOID               TopLevelIrp.
/*218*/    PVOID               DeviceToVerify.
/*21C*/    DWORD               ReadClusterSize.
/*220*/    BOOLEAN             ForwardClusterOnly.
/*221*/    BOOLEAN             DisablePageFaultClustering;
/*222*/    BOOLEAN             DeadThread.
/*223*/    BOOLEAN             Reserved
/*224*/    BOOL                HasTerminated.
/*228*/    ACCESS_MASK         GrantedAccess.
/*22C*/    PEPROCESS           ThreadsProcess.
/*230*/    PVOID               StartAddress.
/*234*/    union
                 {
/*234*/        PVOID           Win32StartAddress;
/*234*/        DWORD           LpcReceivedMessageId.
                 };
/*238*/    BOOLEAN             LpcExitThreadCalled;
/*239*/    BOOLEAN             HardErrorsAreDisabled;
/*23A*/    BOOLEAN             LpcReceivedMsgIdValid;
/*23B*/    BOOLEAN             ActiveImpersonationInfo;
/*23C*/    DWORD               PerformanceCountHigh;
/*240*/    DWORD               d240.
/*244*/    DWORD               d244.
/*248*/    }
     ETHREAD.
 *   PETHREAD.
 **  PPETHREAD;
```

Можно заметить, что в состав структур EPROCESS и ETHREAD входят дополнительные члены помимо тех, информация о которых выводится на экран командами !processfields и !threadfields отладчика. Возможно, для многих будет непонятно, откуда у меня уверенность в том, что команды !processfields и !threadfields выдают неполную информацию об этих структурах? Каким образом я узнал о существовании недокументированных членов? Существуют два пути получить сведения о недокументированных членах структур объектов. Во-первых, можно проследить за тем, как именно система обращается к объекту и его внутренним членам. Во-вторых, можно проанализировать, каким образом происходит создание и инициализация объекта. Воспользовавшись вторым методом, можно узнать фактический размер объекта. Базовой функцией создания объекта является функция ObCreateObject() принадлежащая модулю ntoskrnl.exe. Эта функция выделяет память для размещения заголовка и тела объекта, а также выполняет инициализацию общих парамет-

ров объекта. Вместе с тем функция ObCreateObject() не обладает каким-либо представлением о типе создаваемого объекта, и поэтому вызывающий процесс обязан передать ей в качестве аргумента количество байт, необходимое для размещения тела объекта в памяти. Таким образом, чтобы определить размер объекта, достаточно найти место, где происходит обращение к функции ObCreateObject() для данного объекта. Объекты процессов создаются функцией NtCreateProcess() интерфейса Native API, которая для выполнения всей черновой работы обращается к функции PspCreateProcess(). Именно внутри этой функции можно обнаружить обращение к вызову ObCreateObject(), которому передается требуемый размер 0x288 байт. Исходя из этого размера можно сделать вывод, что в конце структуры EPROCESS необходимо добавить несколько дополнительных полей помимо тех, информация о которых выводится командой !processfields отладчика. Таким образом, размер структуры EPROCESS совпадет с размером, который запрашивается для хранения данного объекта при обращении к функции ObCreateObject(). Точно такую же методику можно применить для того, чтобы определить фактический размер структуры ETHREAD. Функция NtCreateThread() обращается к функции PspCreateThread(), которая, в свою очередь, обращается к ObCreateObject() и передает ей размер объекта, составляющий 0x248 байт.

Список процессов, функционирующих в системе на текущий момент, формируется при помощи членов ActiveProcessLinks структуры EPROCESS. Голова этого списка хранится во внутренней глобальной переменной PsActiveProcessHead, а связанный с ним объект синхронизации типа FAST_MUTEX обладает именем PspActiveProcessMutex. К сожалению, переменная PsActiveProcessHead не экспортируется модулем ntoskrnl.exe, вместе с тем можно получить доступ к переменной PsInitialSystemProcess, которая указывает на структуру EPROCESS, соответствующую процессу System с идентификатором 8. Указатель Blink записи ActiveProcessLinks в составе этой структуры указывает непосредственно на PsActiveProcessHead. Организация ссылок между процессами и потоками показана на рис. 7.3. Очевидно, что эта схема намеренно упрощена: в списке активных процессов присутствуют только две записи. В реальности список процессов, как правило, существенно длиннее. Например, в то время как я пишу данный абзац, диспетчер задач показывает, что на моем компьютере выполняются 36 процессов! Чтобы упростить рисунок, на нем показан список потоков только для одного процесса. Предполагается, что этот процесс обладает только двумя активными потоками.

Если внимательно посмотреть на листинги 7.12 и 7.13, по наличию указателей на объекты типа WIN32_PROCESS и WIN32_THREAD можно сделать вывод о том, что помимо уровня Executive и уровня ядра существует третий уровень объектов процессов и потоков. Структуры объектов третьего уровня описывают процессы и потоки с точки зрения подсистемы Win32. О предназначении некоторых полей структур третьего уровня можно легко догадаться, однако информации об этих структурах, которой я располагаю, явно недостаточно для того, чтобы привести внутреннее строение этих структур в данной книге. Внутреннее строение структур WIN32_PROCESS и WIN32_THREAD — это еще одна пока что непознанная область Windows 2000, исследовать которую только предстоит.

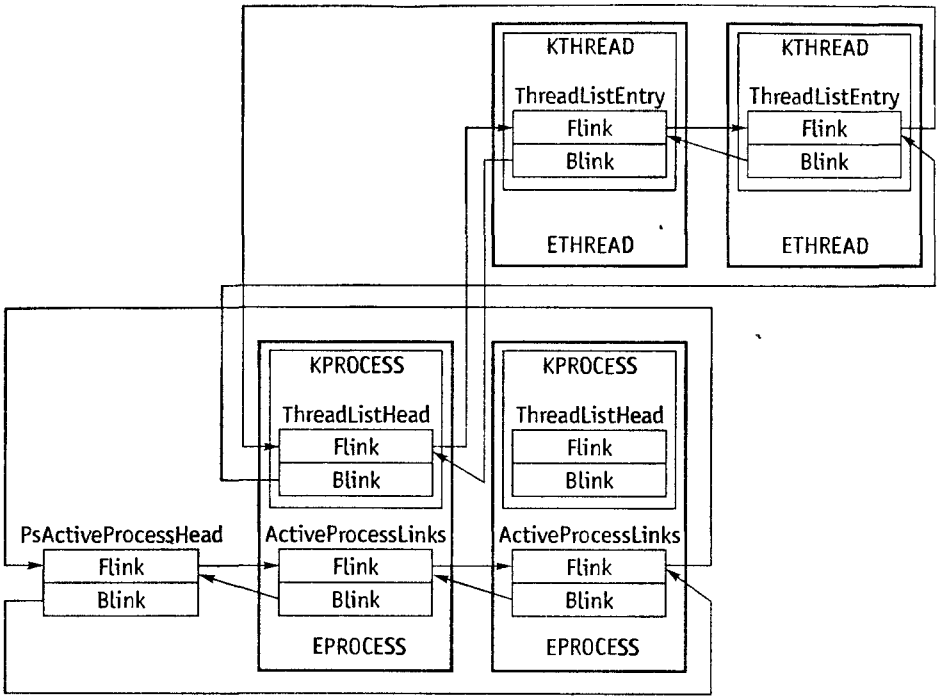

Рис. 7.3. Списки объектов процессов и потоков

Контекст процесса и потока

Когда система выполняет какой-либо код, этот код всегда исполняется в контексте потока, который является частью некоторого процесса. В некоторых ситуациях система нуждается в получении специфичных для процесса или потока данных из текущего контекста. Для этой цели система всегда хранит указатель на текущий поток в структуре KPRCB (Kernel's Processor Control Block). Эта структура определяется в файле ntddk.h. Ее определение приведено в листинге 7.15.

Листинг 7.15. Структура KPRCB (Kernel's Processor Control Block)

```
//  ------------------------------------------------------------------
// базовый адрес 0xFFDFF120

typedef struct _KPRCB // Processor Control Block (Блок управления процессором)
        {
/*000*/    WORD                    MinorVersion;
/*002*/    WORD                    MajorVersion;
/*004*/    struct _KTHREAD         *CurrentThread;
/*008*/    struct _KTHREAD         *NextThread;
/*00C*/    struct _KTHREAD         *IdleThread;
/*010*/    CHAR                    Number;
/*011*/    CHAR                    Reserved;
/*012*/    WORD                    BuildType;
/*014*/    KAFFINITY               SetMember;
```

```
/*018*/    struct _RESTART_BLOCK        *RestartBlock
/*01C*/    }
      KPRCB,
 *    PKPRCB,
 **   PPKPRCB;
```

Структура KPRCB располагается по линейному адресу 0xFFDFF120, а указатель на нее хранится в поле Prcb структуры KPCR (Kernel's Processor Control Region), которая также определена в заголовочном файле ntddk.h (листинг 7.16) и располагается по адресу 0xFFDFF000. Как уже рассказывалось в главе 4, эта структура является важной областью данных, доступ к которой в режиме ядра всегда можно получить при помощи сегмента FS: обращение по адресу FS:0 эквивалентно обращению по линейному адресу DS:0xFFDFF000. А по адресу 0xFFDFF13C, сразу же за структурой KPRCB, располагается структура CONTEXT, в которой система хранит данные, имеющие отношение к управлению центральным процессором на низком уровне (листинг 7.17).

Листинг 7.16. Структура KPCR (Kernel's Processor Control region)

```
//  ----------------------------------------------------------------
// базовый адрес 0xFFDFF000

typedef struct _KPCR // Processor Control Region (Область управления процессором)
     {
/*000*/    NT_TIB                  NtTib,
/*01C*/    struct _KPCR            *SelfPcr;
/*020*/    PKPRCB                  Prcb,
/*024*/    KIRQL                   Irql,
/*028*/    DWORD                   IRR,
/*02C*/    DWORD                   IrrActive;
/*030*/    DWORD                   IDR,
/*034*/    DWORD                   Reserved2;
/*038*/    struct _KIDTENTRY       *IDT;
/*03C*/    struct _KGDTENTRY       *GDT;
/*040*/    struct _KTSS            *TSS,
/*044*/    WORD                    MajorVersion;
/*046*/    WORD                    MinorVersion;
/*048*/    KAFFINITY               SetMember,
/*04C*/    DWORD                   StallScaleFactor;
/*050*/    BYTE                    DebugActive,
/*051*/    BYTE                    Number;
/*054*/    }
      KPCR,
 *    PKPCR,
 **   PPKPCR;
```

Листинг 7.17. Структуры CONTEXT и FLOATING_SAVE_AREA

```
typedef struct _FLOATING_SAVE_AREA
          {
/*000*/    DWORD     ControlWord,
/*004*/    DWORD     StatusWord,
/*008*/    DWORD     TagWord,
/*00C*/    DWORD     ErrorOffset;
/*010*/    DWORD     ErrorSelector,
/*014*/    DWORD     DataOffset,
```

продолжение

Листинг 7.17 (*продолжение*)

```
/*018*/    DWORD     DataSelector;
/*01C*/    BYTE      RegisterArea  [SIZE_OF_80387_REGISTERS];
/*06C*/    DWORD     Cr0NpxState;
/*070*/    }
      FLOATING_SAVE_AREA,
 *    PFLOATING_SAVE_AREA,
 **   PPFLOATING_SAVE_AREA;

//  ---------------------------------------------------------------
// базовый адрес 0xFFDFF13C

#define MAXIMUM_SUPPORTED_EXTENSION 512

typedef struct _CONTEXT
        {
/*000*/    DWORD               ContextFlags;
/*004*/    DWORD               Dr0;
/*008*/    DWORD               Dr1;
/*00C*/    DWORD               Dr2;
/*010*/    DWORD               Dr3;
/*014*/    DWORD               Dr6;
/*018*/    DWORD               Dr7;
/*01C*/    FLOATING_SAVE_AREA  FloatSave;
/*08C*/    DWORD               SegGs;
/*090*/    DWORD               SegFs;
/*094*/    DWORD               SegEs;
/*098*/    DWORD               SegDs;
/*09C*/    DWORD               Edi;
/*0A0*/    DWORD               Esi;
/*0A4*/    DWORD               Ebx;
/*0A8*/    DWORD               Edx;
/*0AC*/    DWORD               Ecx;
/*0B0*/    DWORD               Eax;
/*0B4*/    DWORD               Ebp;
/*0B8*/    DWORD               Eip;
/*0BC*/    DWORD               SegCs;
/*0C0*/    DWORD               EFlags;
/*0C4*/    DWORD               Esp;
/*0C8*/    DWORD               SegSs;
/*0CC*/    BYTE                ExtendedRegisters  [MAXIMUM_SUPPORTED_EXTENSION];
/*2CC*/    }
      CONTEXT,
 *    PCONTEXT,
 **   PPCONTEXT;
```

Изучив листинг 7.15, можно обнаружить, что структура KPRCB содержит три указателя на объекты KTHREAD. Эти три указателя расположены со смещениями 0x004, 0x008 и 0x00C:

1. CurrentThread указывает на объект KTHREAD потока, который исполняется в системе в данный момент. Код ядра очень часто обращается к этому члену структуры.
2. NextThread указывает на объект KTHREAD потока, выполнение которого запланировано после очередного переключения контекста.

3. IdleThread указывает на объект KTHREAD потока холостого хода. *Поток холостого хода* (idle thread) выполняет действия в фоновом режиме в то время, когда ни один из других потоков не готов продолжить работу. Для каждого установленного на компьютере ЦПУ система формирует индивидуально выделенный поток холостого хода. На однопроцессорном компьютере объект потока холостого хода называется P0BootThread. Этот объект является единственным объектом в списке программных потоков процесса PsIdleProcess (процесс холостого хода).

Структура KTHREAD расположена в самом начале структуры ETHREAD, поэтому можно считать, что указатель на KTHREAD также является указателем и на ETHREAD. Это означает, что указатели на KTHREAD и ETHREAD взаимозаменяемы. То же самое можно сказать и об указателях на KPROCESS и EPROCESS.

В режиме ядра ядро Windows 2000 отображает линейный адрес 0xFFDFF000 на адрес 0x00000000 сегмента FS, поэтому система всегда может обнаружить структуры KPCR, KPRCB и CONTEXT по адресам FS:0x0, FS:0x120 и FS:13C. Дизассемблировав код ядра в отладчике, вы можете обнаружить, что система достаточно часто извлекает указатель из ячейки FS:0x124. Как можно заключить из всего вышесказанного, в этой ячейке содержится указатель на объект текущего программного потока. В примере 7.1 показаны данные, выводимые отладчиком ядра по команде u PsGetCurrentProcessId. По этой команде отладчик показывает десять строк дизассемблированного кода, начиная с адреса символа PsGetCurrentProcessId. Функция PsGetCurrentProcessId() извлекает из структуры KPRCB указатель на объект KTHREAD/ETHREAD текущего потока и возвращает значение поля этой структуры, расположенного со смещением 0x1E4. Из листинга 7.14 видно, что по этому смещению располагается поле UniqueProcess структуры Cid типа CLIENT_ID, входящей в состав объекта ETHREAD. В этом поле хранится идентификатор процесса, которому принадлежит текущий поток. Функция PsGetCurrentThreadId() выглядит фактически так же, за исключением того, что она возвращает значение, расположенное в структуре ETHREAD, со смещением 0x1E4. По этому смещению располагается поле UniqueThread структуры CLIENT_ID Cid, то есть идентификатор текущего потока. Строение структуры CLIENT_ID содержится в листинге 2.8 главы 2.

Пример 7.1. Получение ID процесса и потока

```
kd> u PsGetProcessId
u PsGetCurrentProcessId
ntoskrnl!PsGetCurrentProcessId:
8045252a 64a124010000    mov     eax,fs:[00000124]
80452530 8b80e0010000    mov     eax,[eax+0x1e0]
80452536 c3              ret
80452537 cc              int     3
ntoskrnl!PsGetCurrentThreadId:
8045252a 64a124010000    mov     eax,fs:[00000124]
80452530 8b80e4010000    mov     eax,[eax+0x1e4]
80452536 c3              ret
80452537 cc              int     3
```

Иногда система нуждается в получении указателя на объект процесса, которому принадлежит текущий поток. Чтобы получить этот адрес, достаточно прочитать поле Process подструктуры ApcState внутри текущего объекта KTHREAD.

Блоки окружения для потока и процесса

Многих может заинтересовать предназначение полей Teb и Peb структур KTHREAD и EPROCESS. Поле Teb указывает на блок TEB (Thread Environment Block), строение которого показано в листинге 7.18. Название этого блока можно перевести как *блок окружения потока*. В начале блока TEB располагается структура NT_TIB (Thread Information Block), которая определяется в заголовочных файлах winnt.h и ntddk.h, входящих в состав комплектов SDK (Software Development Kit) и DDK (Drivers Development Kit) соответственно. Остальные члены структуры не документированы. Система Windows 2000 создает структуру TEB для каждого объекта потока в системе. В адресном пространстве текущего процесса структуры TEB для потоков, входящих в состав процесса, располагаются по линейным адресам 0x7FFDE000, 0x7FFDD000, 0x7FFDC000 и т. д. с шагом 4 Кбайт (одна страница) для каждого потока. Как уже отмечалось в главе 4, структура TEB текущего потока так же доступна в сегменте FS пользовательского режима. Многие функции ntdll.dll обращаются к TEB текущего потока, читая значение, расположенное по адресу FS:0x18. В этой ячейке размещается поле Self входящей в состав блока TEB структуры NT_TIB. Этот адрес можно считать линейным адресом структуры TEB в линейном 4-гигабайтно адресном пространстве текущего процесса.

Листинг 7.18. Строение блока TEB (Thread Environment Block)

```
// typedef struct _NT_TIB // см. файлы winnt.h / ntddk.h
//         {
// /*000*/ struct  _EXCEPTION_REGISTRATION_RECORD  *ExceptionList.
// /*004*/ PVOID                                    StackBase;
// /*008*/ PVOID                                    StackLimit;
// /*00C*/ PVOID                                    SubSystemTib;
// /*010*/ union
//                 {
// /*010*/ PVOID FiberData;
// /*010*/ ULONG Version;
//                 };
// /*014*/ PVOID                    ArbitraryUserPointer;
// /*018*/ struct _NT_TIB *Self;
// /*01C*/ }
//         NT_TIB.
//    *    PNT_TIB.
//    **   PPNT_TIB;

//  ------------------------------------------------------------------

typedef struct _TEB // базовые адреса 0x7FFDE000, 0x7FFOD000, .

        {
/*000*/   NT_TIB          Tib;
/*01C*/   PVOID           EnvironmentPointer;
/*020*/   CLIENT_ID       Cid;
/*028*/   HANOLE          RpcHandle.
/*02C*/   PPVOID          ThreadLocalStorage;
/*030*/   PPEB            Peb;
/*034*/   DWORD           LastErrorValue;
/*03B*/   }
      TEB.
 *    PTEB.
 **   PPTEB;
```

Подобно потокам, каждому из которых соответствует блок TEB, любой процесс в Windows 2000 обладает собственным *блоком окружения процесса* (Process Environment Block, PEB). Структура блока PEB показана в листинге 7.19. Как видно из листинга, по сравнению с блоком TEB блок PEB обладает куда более сложным внутренним строением. В состав блока PEB входят разнообразные указатели на связанные с этим блоком структуры, большая часть которых не документирована. В листинге 7.19 содержатся приблизительные наброски некоторых таких структур, в которых используются предположительные имена полей. К сожалению, очень многое остается непонятным. Блок PEB располагается по линейному адресу 0x7FFDF00, то есть в первой 4-Кбайтной странице, расположенной за стеком структур TEB процесса. Чтобы получить доступ к PEB, достаточно обратиться к полю Peb блока TEB текущего потока.

Листинг 7.19. Строение блока PEB (Process Environment Block)

```
typedef struct _MODULE_HEADER
        {
/*000*/    DWORD               d000.
/*004*/    DWORD               d004.
/*008*/    LIST_ENTRY          List1;
/*010*/    LIST_ENTRY          List2.
/*018*/    LIST_ENTRY          List3.
/*020*/    }
      MODULE_HEADER.
 *    PMODULE_HEADER.
 **   PPMODULE_HEADER;

//  ------------------------------------------------------------------

typedef struct _PROCESS_MODULE_INFO
        {
/*000*/    DWORD                   Size. // 0x24
/*004*/    MODULE_HEADER           ModuleHeader;
/*024*/    }
      PROCESS_MODULE_INFO.
 *    PPROCESS_MODULE_INFO.
 **   PPPROCESS_MODULE_INFO;

//  ------------------------------------------------------------------
// см. RtlCreateProcessParameters()

typedef struct _PROCESS_PARAMETERS
      {
/*000*/    DWORD               Allocated.
/*004*/    DWORD               Size;
/*008*/    DWORD               Flags;        // бит 0. все указатели нормализованы
/*00C*/    DWORD               Reserved1;
/*010*/    LONG                Console;
/*014*/    DWORD               ProcessGroup.
/*018*/    HANDLE              StdInput;
/*01C*/    HANDLE              StdOutput;
/*020*/    HANDLE              StdError;
/*024*/    UNICODE_STRING      WorkingDirectoryName;
/*02C*/    HANDLE              WorkingDirectoryHandle.
/*030*/    UNICODE_STRING      SearchPath;
```

продолжение

Листинг 7.19 (*продолжение*)

```
/*038*/    UNICODE_STRING     ImagePath;
/*040*/    UNICODE_STRING     CommandLine;
/*048*/    PWORD              Environment;
/*04C*/    DWORD              X;
/*050*/    DWORD              Y;
/*054*/    DWORD              XSize;
/*058*/    DWORD              YSize;
/*05C*/    DWORD              XCountChars;
/*060*/    DWORD              YCountChars;
/*064*/    DWORD              FillAttribute;
/*068*/    DWORD              Flags2;
/*06C*/    WORD               ShowWindow;
/*06E*/    WORD               Reserved2;
/*070*/    UNICODE_STRING     Title;
/*078*/    UNICODE_STRING     Desktop;
/*080*/    UNICODE_STRING     Reserved3;
/*088*/    UNICODE_STRING     Reserved4;
/*090*/    }
     PROCESS_PARAMETERS,
 *   PPROCESS_PARAMETERS,
 **   PPPROCESS_PARAMETERS;

//  ---------------------------------------------------------------

typedef struct _SYSTEM_STRINGS
         {
/*000*/    UNICODE_STRING SystemRoot;          // d:\WINNT
/*008*/    UNICODE_STRING System32Root;        // d:\WINNT\System32
/*010*/    UNICODE_STRING BaseNamedObjects;    // \BaseNamedObjects
/*018*/    }
     SYSTEM_STRINGS,
 *   PSYSTEM_STRINGS,
 **   PPSYSTEM_STRINGS;

//  ---------------------------------------------------------------

typedef struct _TEXT_INFO
         {
/*000*/    PVOID              Reserved;
/*004*/    PSYSTEM_STRINGS    SystemStrings;
/*008*/    }
     TEXT_INFO,
 *   PTEXT_INFO,
 **   PPTEXT_INFO;

//  ---------------------------------------------------------------

typedef struct _PEB // базовый адрес 0x7FFDF000
         {
/*000*/    BOOLEAN            InheritedAddressSpace;
/*001*/    BOOLEAN            ReadImageFileExecOptions;
/*002*/    BOOLEAN            BeingDebugged;
/*003*/    BYTE               b003;
/*004*/    DWORD              d004,
/*008*/    PVOID              SectionBaseAddress;
```

```
/*00C*/    PPROCESS_MODULE_INFO    ProcessModuleInfo;
/*010*/    PPROCESS_PARAMETERS     ProcessParameters;
/*014*/    DWORD                   SubSystemData,
/*018*/    HANDLE                  ProcessHeap,
/*01C*/    PCRITICAL_SECTION       FastPebLock;
/*020*/    PVOID                   AcquireFastPebLock;     // функция
/*024*/    PVOID                   ReleaseFastPebLock;     // функция
/*028*/    DWORD                   d028,
/*02C*/    PPVOID                  User32Dispatch;         // функция
/*030*/    DWORD                   d030;
/*034*/    DWORD                   d034;
/*038*/    DWORD                   d038;
/*03C*/    DWORD                   TlsBitMapSize;          // количество бит
/*040*/    PRTL_BITMAP             TlsBitMap;              // ntdll!TlsBitMap
/*044*/    DWORD                   TlsBitMapData [2];      // 64 бит
/*04C*/    PVOID                   p04C;
/*050*/    PVOID                   p050;
/*054*/    PTEXT_INFO              TextInfo;
/*058*/    PVOID                   InitAnsiCodePageData;
/*05C*/    PVOID                   InitOemCodePageData;
/*060*/    PVOID                   InitUnicodeCaseTableData;
/*064*/    DWORD                   KeNumberProcessors;
/*068*/    DWORD                   NtGlobalFlag;
/*06C*/    DWORD                   d6C,
/*070*/    LARGE_INTEGER           MmCriticalSectionTimeout;
/*078*/    DWORD                   MmHeapSegmentReserve;
/*07C*/    DWORD                   MmHeapSegmentCommit;
/*080*/    DWORD                   MmHeapDeCommitTotalFreeThreshold;
/*084*/    DWORD                   MmHeapDeCommitFreeBlockThreshold;
/*088*/    DWORD                   NumberOfHeaps;
/*08C*/    DWORD                   AvailableHeaps;         // 16, *2 если недостаточно
/*090*/    PHANDLE                 ProcessHeapsListBuffer;
/*094*/    DWORD                   d094;
/*098*/    DWORD                   d098;
/*09C*/    DWORD                   d09C,
/*0A0*/    PCRITICAL_SECTION       LoaderLock;
/*0A4*/    DWORD                   NtMajorVersion;
/*0A8*/    DWORD                   NtMinorVersion;
/*0AC*/    WORD                    NtBuildNumber;
/*0AE*/    WORD                    CmNtCSDVersion;
/*0B0*/    DWORD                   PlatformId;
/*0B4*/    DWORD                   Subsystem;
/*0B8*/    DWORD                   MajorSubsystemVersion;
/*0BC*/    DWORD                   MinorSubsystemVersion;
/*0C0*/    KAFFINITY               AffinityMask;
/*0C4*/    DWORD                   ad0C4 [35];
/*150*/    PVOID                   p150;
/*154*/    DWORD                   ad154 [32];
/*1D4*/    HANDLE                  Win32WindowStation;
/*1D8*/    DWORD                   d1D8;
/*1DC*/    DWORD                   d1DC;
/*1E0*/    PWORD                   CSDVersion;
/*1E4*/    DWORD                   d1E4;
/*1E8*/    }
      PEB,
 *    PPEB,
 **   PPPEB;
```

Доступ к объектам в функционирующей системе

В предыдущих разделах мы рассмотрели достаточно большой объем теоретического материала, настало время научиться применять полученные знания на практике. В качестве практического примера, иллюстрирующего управление объектами Windows 2000 в наиболее удобной форме, я предлагаю рассмотреть программу просмотра объектов ядра. Данная программа демонстрирует, каким образом объекты организованы иерархически и как можно получить доступ к некоторым их свойствам. К сожалению, модуль ntoskrnl.exe не экспортирует несколько важных структур и функций, которые необходимы при разработке приложения просмотра объектов ядра. Это означает, что даже драйверы режима ядра не могут обратиться к этим структурам и функциям — все они зарезервированы системой для внутреннего использования. С другой стороны, в главе 6 был описан механизм, при помощи которого можно получить доступ к неэкспортируемым данным и коду. Работа этого механизма основана на анализе содержимого символьных файлов Windows 2000. Программа просмотра объектов ядра — это подходящий случай, когда можно протестировать эффективность данного подхода. Могу сказать заранее, что технология обращения к внутренностям ядра при помощи символьных файлов, рассмотренная в главе 6, с успехом выдержала это испытание. Исполняемый файл w2k_obj.exe программы просмотра объектов Windows 2000 в комплекте с полным исходным кодом содержится в каталоге \src\w2k_obj на прилагаемом к книге компакт-диске. Однако наиболее интересная часть кода находится не в файле w2k_obj.c, а внутри библиотеки w2k_call.dll, которая рассматривалась в главе 6. По этой причине многие рассматриваемые далее фрагменты исходного кода позаимствованы из файла w2k_call.c.

Последовательный перебор записей каталога объектов

Возможно, вам известно о существовании небольшой утилиты objdir.exe, входящей в состав Windows 2000 DDK, которая располагается в каталоге \ntddk\bin. Утилита objdir.exe получает сведения из каталога объектов Windows 2000 при помощи недокументированной функции Native API под названием NtQueryDirectoryObject(), экспортируемой библиотекой ntdll.dll. Разработанная мной программа w2k_obj.exe использует другой подход: она напрямую обращается к каталогу объектов Windows и читает данные из замыкающих объектов каталога. Это звучит пугающе, однако на самом деле ничего страшного в этом нет. Лучшим доказательством безопасности такого подхода является то обстоятельство, что программа w2k_obj.exe отлично работает как в среде Windows 2000, так и в среде Windows NT 4.0. Иными словами, в ней нет ни единой строчки, критичной к версии ОС. Следует отметить, что некоторые тонкие отличия между каталогами объектов Windows NT 4.0 и Windows 2000 все же существуют, однако базовая модель каталога в обеих ОС одна и та же. Рассмотрев работу программы, которая работает напрямую со структурами объектов, не используя при этом каких-либо высокоуровневых функций API,

мы убедимся в том, что изложенные в предыдущих разделах сведения о внутренних структурах Windows 2000 соответствуют действительности и их действительно можно с успехом применять при разработке собственных программ.

Прежде чем приступать к работе с глобальными структурами данных системы, важно заблокировать их. В противном случае в ходе выполнения какой-либо операции с данными эти данные могут быть изменены потоком, который выполняется параллельно с текущим. В результате могут быть прочитаны неправильные данные или приложение получит указатель, который никуда не указывает. Для разнообразных внутренних данных, с которыми работает Windows 2000, эта система поддерживает обширный набор объектов синхронизации, позволяющих блокировать доступ к тому или иному объекту самым причудливым образом. Проблема заключается в том, что большая часть этих объектов блокирования не экспортируется. Несмотря на то что драйвер режима ядра может выполнять многие действия, которые запрещены для приложений пользовательского режима, все же он не может работать со структурами данных, которые не экспортируются ядром. Вместе с тем расширенный интерфейс обращения к ядру, рассмотренный в главе 6 и реализованный в библиотеке w2k_call.dll, позволяет сделать невозможное возможным. Для этого он извлекает адреса внутренних символов операционной системы из символьных файлов ОС. Модуль w2k_call.dll экспортирует три важные функции, благодаря которым работа с каталогом объектов ядра становится возможной:

1. __ObpRootDirectoryMutex() возвращает адрес объекта синхронизации ERESOURCE, при помощи которого синхронизируется доступ к каталогу объектов как к единому целому.
2. __ObpRootDirectoryObject() возвращает указатель на структуру OBJECT_DIRECTORY, которая описывает корневой узел каталога объектов.
3. __ObpTypeDirectoryObject() возвращает указатель на структуру типа OBJECT_DIRECTORY, которая описывает подкаталог \ObjectTypes каталога объектов.

При работе с указателями на объекты ядра приложение должно вести себя чрезвычайно осторожно, особенно при блокировании доступа к каталогу. Если разблокирование каталога выполнить некорректно, система окажется со связанными руками и не сможет выполнить даже самые простые задачи.

Несмотря на то что объект блокирования каталога называется ObpRootDirectoryMutex, на самом деле он не является мьютексом в строгом смысле этого слова. В действительности тип этого объекта не KMUTEX, а ERESOURCE, и поэтому для захвата этого объекта следует использовать одну из функций: ExAcquireResourceExclusiveLite() или ExAcquireResourceSharedLite(). Обратите внимание на наличие суффикса «Lite» — в среде Windows 2000 или Windows NT 4.0 никогда не следует использовать в отношении объектов типа ERESOURCE функцию ExAcquireResourceExclusive() или ExAcquireResourceShared(). Дело в том что, в отличие от Windows NT 3.x в новых версиях Windows структура ERESOURCE претерпела существенные изменения, а функции, в именах которых отсутствует суффикс «Lite», предназначены для работы со старой версией структуры ERESOURCE, определение которой включено в состав файла w2k_def.h как ERESOURCE_OLD (см. приложение В). Функция, вы-

полняющая действие, обратное функции ExAcquireResource*Lite(), называется ExReleaseResourceLite(). Вместо этой функции ни в коем случае не следует использовать функцию ExReleaseResource(), так как последняя предназначена для работы со старой версией объекта синхронизации.

Разработанная мной программа просмотра каталога объектов работает очень просто: она блокирует каталог объектов, делает снимок всех объектов, которые можно обнаружить в каталоге и после разблокирования каталога отображает извлеченные из каталога данные на экране. Подобный подход предусматривает минимальное влияние на работу системы, так как, быстро получив из каталога все данные, приложение разблокирует его, а затем не спеша может выполнить отображение этих данных, а также в случае необходимости любые другие процедуры, связанные с их обработкой. Для получения снимка каталога объектов требуются чрезвычайно глубокие знания о строении структур системных объектов. Таким образом, изучив функционирование предлагаемой мною программы, читатель сможет убедиться в том, что содержащиеся в данной главе теоретические сведения соответствуют действительности и их можно с успехом использовать на практике. Задачу получения снимка каталога объектов можно разделить на две подзадачи:

1. Копирование *структуры* дерева каталога объектов. Эта задача подразумевает копирование и связывание в логическую иерархию нескольких структур OBJECT_DIRECTORY, каждая из которых представляет собой отдельный узел дерева каталога, который не является замыкающим.
2. Копирование *содержимого* дерева каталога объектов. Эта задача подразумевает копирование структуры OBJECT_HEADER, а также всех остальных структур каждого замыкающего узла дерева каталога.

Первая подзадача выполняется функцией w2kDirectoryOpen(), исходный код которой представлен в листинге 7.20. Эта функция блокирует каталог и последовательно перебирает все дочерние узлы переданного ей в виде аргумента объекта OBJECT_DIRECTORY. Чтобы получить полные сведения о структуре всего дерева каталога, обращение к этой функции должно выполняться рекурсивно для каждой записи каталога, которая является объектом типа OBJECT_DIRECTORY. Не забывайте о том, что каждый узел каталога содержит хэш-таблицу, в которой может быть до 37 записей. Каждая запись хэш-таблицы может содержать в себе указатель на список, в котором может быть произвольное число элементов. Таким образом, для последовательного перебора записей каталога требуется организовать два вложенных цикла: внешний цикл просматривает 37 слотов хэш-таблицы в поисках элементов, значение которых отличается от NULL, а внутренний цикл последовательно перебирает все записи списка. Вот и все, что делает функция w2kDirectoryOpen(). Полученные в результате ее работы данные структурно эквивалентны изначальной модели каталога объектов, однако функция размещает эти данные в памяти, которая доступна из пользовательского режима. Основное копирование, включая автоматическое выделение памяти, выполняется чрезвычайно мощной функцией w2kSpyClone(), которая также экспортируется модулем w2k_call.dll (листинг 6.30). Функция w2kDirectoryClose() выполняет процедуру, обратную функции w2kDirectoryOpen(), иначе говоря, она освобождает память, занятую клонированными блоками.

Листинг 7.20. Функции w2kDirectoryOpen() и w2kDirectoryClose()

```
POBJECT_DIRECTORY  WINAPI
w2kDirectoryOpen  (POBJECT_DIRECTORY  pDir)
 {
 DWORD                                  i,
 PERESOURCE                        pLock;
 PPOBJECT_DIRECTORY_ENTRYppEntry;
 PDBJECT_DIRECTDRY             pDir1 = NULL;

 if (((pLock = __ObpRootDirectoryMutex ()) != NULL) &&
     _ExAcquireResourceExclusiveLite  (pLock,  TRUE))
     {
     if ((pDir1 = w2kSpyClone (pDir, OBJECT_DIRECTORY_)) != NULL)
          {
          for (i = 0; i < OBJECT_HASH_TABLE_SIZE, i++)
               {
               ppEntry = pDir1->HashTable + i;

               while (*ppEntry != NULL)
                    {
                    if ((*ppEntry =
                              w2kSpyClone  (*ppEntry,
                                        OBJECT_DIRECTORY_ENTRY_))
                         != NULL)
                         {
                         (*ppEntry)->Object =
                              w2kObjectOpen  ((*ppEntry)->Object),

                         ppEntry = &(*ppEntry)->NextEntry;
                         }
                    }
               }
          }
     _ExReleaseResourceLite  (pLock);
     }
 return pDir1;
 }

//  ---------------------------------------------------------------

PDBJECT_DIRECTORY  WINAPI
w2kDirectoryClose  (POBJECT_DIRECTORY  pDir)
 {
 POBJECT_DIRECTORY_ENTRY      pEntry, pEntry1,
 DWORD                                  i;

 if (pDir != NULL)
     {
     for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
          {
          for (pEntry  = pDir->HashTable [i];
               pEntry != NULL,
               pEntry  = pEntry1)
              {
              pEntry1 = pEntry->NextEntry;
```

продолжение

Листинг 7.20 (*продолжение*)

```
                w2kObjectClose    (pEntry->Object).
                w2kMemoryDestroy  (pEntry);
                }
            }
        w2kMemoryDestroy  (pDir);
        }
    return NULL;
    }
```

Если внимательней изучить листинг 7.20, можно обнаружить, что функция w2kDirectoryOpen() обращается к вызову w2kObjectOpen, а функция w2kDirectoryClose() — соответственно к вызову w2kObjectClose(). Функция w2kObjectOpen() выполняет вторую часть процедуры клонирования каталога объектов: она копирует в память пользовательского режима замыкающий объект каталога. Следует учитывать, что эта функция не выполняет полноценного копирования всего содержимого объекта, так как для выполнения этой задачи потребовалось бы идентифицировать тип каждого объекта, а затем копировать нужное количество байт из тела объекта в пользовательскую память. Вместо этого данная функция копирует полное содержимое заголовка объекта, включая большую часть связанных с ним структур, после чего формирует ложный объект, содержащий указатели на реальное тело объекта и на различные части копии заголовка объекта. Структуры данных, формируемые и инициализируемые функцией w2kObjectOpen(), показаны в листинге 7.21. Структура W2K_OBJECT_FRAME — это монолитный блок данных, содержащий копию заголовка объекта и ложное тело объекта. Ложное тело объекта представлено структурой W2K_OBJECT, которая представляет собой набор указателей на поля структуры W2K_OBJECT_FRAME. Функция w2kObjectOpen() выделяет память для структуры W2K_OBJECT_FRAME, инициализирует ее данными изначального объекта и возвращает указатель на поле Object этой структуры. Если вы вспомните приведенное ранее в данной главе описание тел и заголовков объектов, вы заметите, что структура W2K_OBJECT_FRAME напоминает структуру реального объекта ядра Windows 2000. И в самом деле, в ее состав входят все поля, которыми обладает реальный объект ядра. Приложение может работать с этими полями точно так же, как система работает с объектами в режиме ядра. Для этого используются смещения и флаги, расположенные в структуре OBJECT_HEADER.

Листинг 7.21. Структуры, представляющие собой клон объекта

```
typedef struct _W2K_OBJECT
 {
 POBJECT                pObject.
 POBJECT_HEADER         pHeader.
 POBJECT_CREATOR_INFO   pCreatorInfo.
 POBJECT_NAME           pName.
 POBJECT_HANDLE_DB      pHandleDB.
 POBJECT_QUOTA_CHARGES  pQuotaCharges;
 POBJECT_TYPE           pType.
 PQUOTA_BLOCK           pQuotaBlock.
 POBJECT_CREATE_INFO    pCreateInfo;
 PWORD                  pwName.
 PWORD                  pwType.
 }
```

```
 W2K_OBJECT, *PW2K_OBJECT, **PPW2K_OBJECT,

#define W2K_OBJECT_ sizeof (W2K_OBJECT)

//  ---------------------------------------------------------------

typedef struct _W2K_OBJECT_FRAME
 {
 OBJECT_QUOTA_CHARGES        QuotaCharges,
 OBJECT_HANDLE_DB            HandleDB,
 OBJECT_NAME                 Name,
 OBJECT_CREATOR_INFO         CreatorInfo,
 OBJECT_HEADER               Header;
 W2K_OBJECT                  Object,
 OBJECT_TYPE                 Type,
 QUOTA_BLOCK                 QuotaBlock,
 OBJECT_CREATE_INFO          CreateInfo,
 WORD                        Buffer [],
 }
 W2K_OBJECT_FRAME, *PW2K_OBJECT_FRAME, **PPW2K_OBJECT_FRAME;

#define W2K_OBJECT_FRAME_ sizeof (W2K_OBJECT_FRAME)
#define W2K_OBJECT_FRAME__(_n) (W2K_OBJECT_FRAME_ + ((_n) * WORD_))
```

Мне не хотелось бы слишком глубоко вдаваться в обсуждение подробностей работы функции w2kOpenObject() и всех используемых ею функций. Чтобы проиллюстрировать методики, используемые этими функциями, в листинге 7.22 я привожу исходный текст трех функций, осуществляющих клонирование частей объекта. Функция w2kObjectHeader() создаст копию структуры OBJECT_HEADER, являющейся базовой составляющей заголовка объекта, а функции w2kObjectCreatorInfo() и w2kObjectName() копируют входящие в состав заголовка структуры OBJECT_CREATOR_INFO и OBJECT_NAME в случае, если эти структуры присутствуют в заголовке объекта. И снова в качестве базовой функции конирования используется функция w2kSpyClone(). Дополнительные примеры на эту тему содержатся в исходном файле w2k_call.c на прилагаемом к книге компакт-диске.

Листинг 7.22. Вспомогательные функции, используемые для клонирования объекта

```
POBJECT_HEADER WINAPI
w2kObjectHeader (POBJECT pObject)
 {
 DWORD                  dOffset = OBJECT_HEADER_,
 POBJECT_HEADER pHeader = NULL,

 if (pObject != NULL)
     {
     pHeader = w2kSpyClone (BACK (pObject, dOffset),
                            dOffset);
     }
 return pHeader;
 }

//  ---------------------------------------------------------------

POBJECT_CREATOR_INFO  WINAPI
w2kObjectCreatorInfo (POBJECT_HEADER    pHeader,
                      POBJECT           pObject)
```

продолжение

Листинг 7.22 (*продолжение*)

```
 {
 DWORD                  dOffset;
 POBJECT_CREATOR_INFO pCreatorInfo = NULL;

 if ((pHeader != NULL) && (pObject != NULL) &&
     (pHeader->ObjectFlags & OB_FLAG_CREATOR_INFO))
     {
     dOffset = OBJECT_CREATOR_INFO_ + OBJECT_HEADER_;

     pCreatorInfo = w2kSpyClone (BACK (pObject, dOffset),
                                 OBJECT_CREATOR_INFO_);
     }
 return pCreatorInfo;
 }

//  ------------------------------------------------------------

POBJECT_NAME WINAPI
w2kObjectName   (POBJECT_HEADER  pHeader,
                 POBJECT         pObject)
 {
 DWORD          dOffset;
 POBJECT_NAME pName = NULL;

 if ((pHeader != NULL) && (pObject != NULL) &&
     (dOffset = pHeader->NameOffset))
     {
     dOffset += OBJECT_HEADER_;

     pName = w2kSpyClone (BACK (pObject, dOffset),
                          OBJECT_NAME_);
     }
 return pName;
 }
```

Итак, функция w2kDirectoryOpen() принимает указатель на реальный объект OBJECT_DIRECTORY и возвращает копию, содержащую указатели на структуры W2K_OBJECT, в каждой из которых содержится указатель на тело реального объекта ядра. Программа просмотра объектов обращается к этой функции несколько раз — по одному разу для каждого уровня каталога, который она отображает. В листинге 7.23 содержится сильно сокращенная версия исходного кода программы просмотра объектов ядра. Чтобы получить сокращенную версию, я убрал из исходного кода все фрагменты, не имеющие прямого отношения к процедуре просмотра каталога объектов. Благодаря этому читатель может сконцентрировать внимание на основных процедурах, связанных с просмотром каталога. Полная версия исходного кода содержится в файле w2k_obj.c. Этот файл изобилует множеством дополнительных фрагментов, которые делают основную функциональную структуру приложения менее понятной. Функция, расположенная на самом верхнем уровне, называется DisplayObjects(). При помощи вызова __ObpRootDirectoryObject(), входящего в состав модуля w2k_call.dll, эта функция получает указатель на корневой объект каталога и передает этот указатель вызову DisplayObject(). Функция DisplayObject() выводит на экран имя и тип объекта, а затем, если объект принадле-

жит к категории OBJECT_DIRECTORY, вызывает саму себя рекурсивно. Для каждого более низкого уровня объектов функция DisplayObject() добавляет отступ из трех пробелов. Для полноты картины я добавил код функций из листинга 7.23 в файл w2k_obj.c, размещенный на прилагаемом к книге компакт-диске. Данные функции размещаются в этом файле под заголовком POOR MAN'S OBJECT BROWSER (Программа просмотра объектов для бедных). Основная программа не обращается к этому коду, однако он является вполне функциональным.

Листинг 7.23. Очень простая программа просмотра объектов

```
VOID WINAPI _DisplayObject  (PW2K_OBJECT  pObject,
                             DWORD        dLevel)
 {
 POBJECT_DIRECTORY        pDir;
 POBJECT_DIRECTORY_ENTRY  pEntry;
 DWORD                    i;

 for (i = 0; i < dLevel; i++) printf (L"   ");

 _printf (L"%+.-16s%s\r\n", pObject->pwType, pObject->pwName);

 if ((!lstrcmp (pObject->pwType, L"Directory")) &&
     ((pDir = w2kDirectoryOpen (pObject->pObject)) != NULL))
     {
     for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
          {
          for (pEntry  = pDir->HashTable [i];
                    pEntry != NULL;
                    pEntry  = pEntry->NextEntry)
               {
               _DisplayObject (pEntry->Object, dLevel+1);
               }
          }
     w2kDirectoryClose (pDir);
     }
 return;
 }

//  ---------------------------------------------------------------

VOID WINAPI _DisplayObjects (VOID)
 {
 PW2K_OBJECT pObject;

 if ((pObject = w2kObjectOpen (__ObpRootDirectoryObject ()))
     != NULL)
     {
     _DisplayObject (pObject, 0);
     w2kObjectClose (pObject);
     }
 return;
 }
```

В примере 7.2 показаны некоторые наиболее характерные фрагменты листинга каталога объектов Windows 2000, выведенного на экран программой, исходный код которой содержится в листинге 7.23. Можно заметить, что каталог включает в

себя несколько основных разделов. Например, подкаталог \BaseNamedObjects содержит в себе именованные объекты, доступ к которым, как правило, можно получить из разных процессов. Любой из этих объектов можно открыть, идентифицировав его по имени. В подкаталоге \ObjectTypes содержатся все 27 объектов типов (объект типа — это структура OBJECT_TYPE, определение которой содержится в листинге 7.9), поддерживаемых системой. Объекты типов операционной системы Windows 2000 перечислены в табл. 7.4.

Пример 7.2. Фрагменты каталога объектов Windows 2000

```
Directory       \
 Directory      ArcName
      SymbolicLink    multi(0)disk(0)rdisk(0)
      SymbolicLink    multi(0)disk(0)rdisk(1)
      SymbolicLink    multi(0)disk(0)rdisk(1)partition(1)
      SymbolicLink    multi(0)disk(0)rdisk(0)partition(1)
      SymbolicLink    multi(0)disk(0)fdisk(0)
      SymbolicLink    multi(0)disk(0)rdisk(0)partition(2)
 Device         Ntfs
 Port           SeLsaCommandPort
 Key            REGISTRY
 Port           XactSrvLpcPort
 Port           DbgUiApiPort
 Directory      NLS
      Section         NlsSectionCP874
      Section         NlsSectionCP950
      Section         NlsSectionCP20290
      Section         NlsSectionCP1255c_1255.nls

 Directory      BaseNamedObjects
      Section         DfSharedHeapE445BB
      Section         DFMap0 14765686
      Mutant          ZonesCacheCounterMutex
      Section         DFMap0 14364447
      Event           WINMGMT_COREDLL_UNLOADED
      Mutant          MCICDA_DeviceCritSec_19
      Event           AgentTowkssvcEvent
      Event           userenv  Machine Group Policy has been applied
      SymbolicLink    Local
      Section         DFMap0 15555297
      Section         DfSharedHeapED2256
      Section         DfSharedHeapE8F975
      Section         DFMap0 15232696
      Section         DFMap0 15170325
      Event           Shell_NotificationCallbacksOutstanding
      Section         DFMap0 14364985
      Event           SETTermEvent
      Event           winlogon   User GPO Event 112121

 Directory      ObjectTypes
      Type            Directory
      Type            Mutant
      Type            Thread
      Type            Controller
      Type            Profile
      Type            Event
```

```
      Type            Type
      Type            Section
      Type            EventPair
      Type            SymbolicLink
      Type            Desktop
      Type            Timer
      Type            File
      Type            WindowStation
      Type            Driver
      Type            WmiGuid
      Type            Device
      Type            Token
      Type            IoCompletion
      Type            Process
      Type            Adapter
      Type            Key
      Type            Job
      Type            WaitablePort
      Type            Port
      Type            Callback
      Type            Semaphore
 Directory       Security
      Event           TRKWKS_EVENT
      WaitablePort    TRKWKS_PORT
      Event           LSA_AUTHENTICATION_INITIALIZED
      Event           NetworkProviderLoad
```

Исходный код полноценной программы просмотра объектов (w2k_obj.exe) не только отображает дерево каталогов в более удобной визуальной форме, но также предоставляет пользователю сведения о дополнительных свойствах объектов, а кроме того, поддерживает фильтрацию объектов по типам. В примере 7.3 показан перечень ключей командной строки программы w2k_obj.exe.

Пример 7.3. Электронная подсказка программы w2k_obj.exe

```
// w2k_obj.exe
// SBS Windows 2000 Object Browser V1.00
// 08.27.2000 Sven B. Schreiber
// sbs@orgon.com

Usage:  w2k_obj [+-atf] [<type>] [<#>|-1] [/root] [/type]

+a  -a          show/hide object adresses          (default  -a)
+t  -t          show/hide object type names        (default  +t)
+f  -f          show/hide object flags             (default  -f)
<type>          show <type> objects only           (default  *)
<#>             show <#> directory levels          (default  1)
 -1             show all directory levels
/root           show ObpRootDirectoryObject tree
/type           show ObpTypeDirectoryObject tree

Example:  w2k_obj +atf *port 2 /root

This command displays all Port and WaitablePort objects
starting in the root and scanning two directory levels.
Each line includes address, type, and flag information.
```

В примере 7.4 показан вывод команды w2k_obj +atf *port 2 /root, которая приводится в электронной подсказке в качестве примера. Эта команда выводит на экран информацию только об объектах типа Port и WaitablePort (так как в командной строке указан ключ фильтрации типа *port). Для каждого такого объекта отображается адрес тела объекта, имя типа и набор флагов. При этом программа ограничивает просмотр дерева каталога только двумя уровнями.

Пример 7.4. Вывод команды w2k_obj +atf *port 2 /root

```
Root directory contents  (2 levels shown)

_____

 8149CDD0 Directory_____ <32> \
> |_ E26A0540 Port_________ <24> SeLsaCommandPort
> |_ E130cc20 Port_________ <24> XactSrvLpcPort
> |_ E13E2380 Port_________ <24> DbgUiApiPort
> |_ E13E4BA0 Port_________ <26> SeRmCommandPort
> |_ E26A9D20 Port_________ <24> LsaAuthenticationPort
> |_ E13E4CA0 Port_________ <24> DbgSsApiPort
> |_ E13E3260 Port_________ <24> SmApiPort
> |_ E2707680 Port_________ <24> ErrorLogPort
   |_ 81499B70 Directory_____ <32> \ArcName
   |_ 812FDB60 Directory_____ <32> \NLS
   |_ 814940B0 Directory_____ <32> \Driver
   |_ 81490B30 Directory_____ <32> \WmiGuid
   |_ 81499A90 Directory_____ <32> \Device
   |    |_ 814AEA90 Directory_____ <32> \Device\DmControl
   |    |_ 814AE4F0 Directory_____ <32> \Device\HarddiskDmVolumes
   |    |_ 8148BE50 Directory_____ <32> \Device\Ide
   |    |_ 814AB3D0 Directory_____ <32> \Device\Harddisk0
   |    |_ 814852F0 Directory_____ <32> \Device\Harddisk1
   |    |_ 814A9F50 Directory_____ <32> \Device\WinDfs
   |    \_ 814AB030 Directory_____ <32> \Device\Scsi
   |_ 81319030 Directory_____ <30> \Windows
> |    |_ E2615520 Port_________ <24> SbApiPort
> |    |_ E260E1A0 Port_________ <24> ApiPort
> |    \_ 812FC810 Directory_____ <32> \Windows\WindowStations
   |_ 81319150 Directory_____ <30> \RPC Control
> |    |_ E26B6A20 Port_________ <24> tapsrvlpc
> |    |_ E3228440 Port_________ <24> OLE3c
> |    |_ E269F360 Port_________ <24> spoolss
> |    |_ E269B6E0 Port_________ <24> OLE2
> |    |_ E2C96C60 Port_________ <24> OLE3f
> |    |_ E1306BC0 Port_________ <24> OLE3
> |    |_ E269BD20 Port_________ <24> LRPC0000021c.00000001
> |    |_ E276D520 Port_________ <24> OLE5
> |    |_ E2699D40 Port_________ <24> OLE6
> |    |_ E2697C00 Port_________ <24> OLE7
> |    |_ E26F0AE0 Port_________ <24> ntsvcs
> |    |_ E26B6B20 Port_________ <24> policyagent
> |    |_ E2814CA0 Port_________ <24> OLEa
> |    |_ E29DC3C0 Port_________ <24> OLEb
> |    |_ E304C8A0 Port_________ <24> OLE40
> |    |_ E3165660 Port_________ <24> OLE41
> |    |_ E26979A0 Port_________ <24> epmapper
> |    |_ E13069A0 Port_________ <24> senssvc
```

```
> |   \_ E2C8D040 Port_________ <24> OLE42
  |_ 812FD030 Directory_____ <30> \BaseNamedObjects
  |   \_ 812FDF50 Directory_____ <30> \BaseNamedObjects\Restricted
  |_ 8149CBD0 Directory_____ <32> \??
  |_ 814B5030 Directory_____ <32> \FileSystem
  |_ 8149CCB0 Directory_____ <32> \ObjectTypes
  |_ 81499C50 Directory_____ <32> \Security
> |   \_ 8121EB20 WaitablePort__ <24> TRKWKS_PORT
  |_ 8149B2D0 Directory_____ <32> \Callback
  \_ 81446E90 Directory_____ <30> \KnownDlls
```

Обратите внимание на то, что объекты типа Directory всегда отображаются в составе списка объектов, даже в случае, когда они не соответствуют маске типа объектов. Если бы это было не так, было бы непонятно, в каком из разделов каталога содержится тот или иной интересующий пользователя объект. Чтобы отличить объекты, имена типов которых соответствуют введенной пользователем маске, от объектов типа Directory, напротив объектов, не являющихся объектами Directory, в начале строки выводится символ >.

Что дальше?

О внутренностях операционной системы Windows 2000 можно рассказывать бесконечно, однако количество страниц в любой, даже очень толстой книге, ограничено, поэтому рано или поздно наступает момент, когда надо остановиться. Семь глав этой книги нельзя назвать легким чтением, однако я надеюсь, что мне удалось произвести на читателя неизгладимое впечатление. Если теперь, после прочтения моей книги, вы смотрите на Windows 2000 другими глазами, я считаю, что мне удалось достичь поставленной цели. Если вы являетесь разработчиком средств программирования или отладки, надеюсь, что материал данной книги поможет вам добавить в ваш продукт возможности, которые отсутствуют в продуктах ваших конкурентов. Если же вы разрабатываете программное обеспечение другого характера, я уверен, что знание внутреннего устройства Windows 2000 поможет вам написать более эффективный код, который более эффективно использует возможности этой операционной системы. Я также надеюсь, что данная книга послужит источником вдохновения для множества пытливых умов и что в дальнейшем благодаря энтузиазму и любознательности читателей удастся пролить свет на многие остающиеся непознанными разделы ядра Windows 2000. Никогда в прошлом я не считал, что рассмотрение операционной системы в качестве черного ящика является хорошим подходом к программированию. И сейчас моя точка зрения по этому вопросу остается неизменной.

Команды Kernel Debugger

В таблице приведен краткий справочник по командам консольного интерфейса отладчика Windows 2000 Kernel Debugger, описанного в главе 1.

Таблица А.1. Встроенные команды Kernel Debugger

Команда	Описание
A[<адрес>]	Ассемблировать
BA[#]<e|r|w|i><1|2|4><адрес>	Точка останова по адресу
BC[<bp>]	Очистить точку(и) останова
BD[<bp>]	Отключить точку(и) останова
BE[<bp>]	Включить точку(и) останова
BL[<bp>]	Вывести точку(и) останова
BP[#]<адрес>	Установить точку(и) останова
C<диапазон><адрес>	Сравнить память
D[тип] [<диапазон>]	Сделать снимок памяти
E[тип] <адрес>[<список>]	Ввод
F<диапазон><список>	Заполнить память
G[=<адрес>[<адрес>...]]	Перейти по адресу
I<тип><порт>	Прочитать из порта ввода-вывода
J<выражение>['']cmd1['];['']cmd2[']	Условное выполнение
K[B]<счетчик>	Отследить стек
KB=<база><стек><ip>	Отследить стек, начиная с определенного состояния
L{+|-}[lost*]	Управляет исходными параметрами
LN<выражение>	Вывести ближайшие идентификаторы
LS[.][<первая>][,<счетчик>]	Вывести строки исходного файла
LSA<адрес>[,<первая>][,<счетчик>]	Вывести строки исходного файла по адресу
LSC	Показать текущие исходный файл и строку
LSF[-]<файл>	Загрузить или выгрузить исходный файл для просмотра
M<диапазон><адрес>	Переместить память
N[<основание>]	Установить/показать основание системы счисления

Команда	Описание
P[R][=<адрес>][<значение>]	Шаг программы
Q	Выйти из отладчика
#R	Снимок регистров нескольких процессоров
R[F][L][M<выражение>][[<регистр>[=<выражение>]]]	Получить/установить значение регистра/флага
Rm[?][<выражение>]	Управляет видом подсказки для маски выходных данных регистров
S<диапазон><список>	Искать в памяти
SS<n | a | w>	Установить суффикс идентификатора
SX[e|d[<событие>|*|<выражение>]]]	Исключение
T[R][=<адрес>][<выражение>]	Отслеживать
U[<диапазон>]	Дизассемблировать
O<тип><порт><выражение>	Записать в порт ввода-вывода
X[<*|модуль>!]<*|идентификатор>	Проверить идентификаторы
.cache[размер]	Установить размер кэша виртуальной памяти
.logopen[<файл>]	Открыть новый файл журнала
.logappend[<файл>]	Добавить к файлу журнала
.logclose	Закрыть файл журнала
.reboot	Перезагрузить компьютер
.reload	Загрузить идентификаторы заново
~<процессор>	Сменить текущий процессор
?<выражение>	Показать выражение
#<строка>[адрес]	Поиск строки в дизассемблированном коде
$<<имя файла>	Прочитать входные данные из командного файла

Таблица А.2. Типы аргументов команд из табл. А.1

Аргумент	Описание
<адрес>	#<16-битный адрес [seg:]address защищенного режима> &<адрес [seg:]address в режиме VB6>
<событие>	ct, et, ld, av, cc
<выражение>	операции: + - * / not by wo dw poi mod(%) and(&) xor(^) or(|) hi low операнды: число в текущей системе счисления, открытый идентификатор, <регистр>
<флаг>	iopl, of, df, if, tf, sf, zf, af, pf, cf
<список>	<байт>[<байт>...]
<шаблон>	[(nt | <имя dll>)!]<имя переменной> (<имя переменной> может содержать ? и *)
<основание>	8, 10, 16
<регистр>	[e]ax, [e]bx, [e]cx, [e]dx, [e]si, [e]di, [e]bp, [e]sp, [e]ip, [e]fl al, ah, bl, bh, cl, ch, dl, dh, cs, ds, es, fs, gs, ss cr0, cr2, cr3, cr4, dr0, dr1, dr2, dr3, dr6, dr7 gdtr, gdtl, idtr, idtl, tr, ldtr
<тип>	b (BYTE) w (WORD) d[s] (DWORD [с идентификаторами]) q (QWORD) f (FLOAT) D (DOUBLE) a (ASCII) c (DWORD и CHAR) u (Unicode) s|S (строка ANSI/Unicode) l (список)

Таблица А.3. Прямые команды, экспортируемые kdextx86.dll

Команда	Описание
!acl<Address>[флаги]	Показать ACL
!apic[база]	Сделать снимок памяти локального APIC
!arbiter[флаги]	Показать все арбитры и диапазоны арбитража флаги: 1 арбитры ввода-вывода 2 арбитры памяти 4 арбитры IRQ 8 арбитры DMA 10 арбитры номеров шин
!arblist<адрес>[флаги]	Сделать снимок памяти ресурсов, арбитраж которых проводится флаги: 1 включить информацию об интерфейсе и разъеме каждого устройства
!bugdump	Показать данные памяти для проверки ошибок
!bushnd	Сделать снимок памяти списка структур HAL «BUS HANDLER»
!bushnd<адрес>	Сделать снимок памяти структуры HAL «BUS HANDLER» для обрабатываемого адреса <адрес>
!ca<адрес>[флаги]	Сделать снимок памяти управляющей области раздела
!callback<адрес>[номер]	Сделать снимок памяти кадров обратного вызова для указанного потока
!calldata<имя таблицы>	Сделать снимок памяти хэш-таблицы данных вызова
!cbreg<BaseAddr> | %%<PhyAddr>	Сделать снимок памяти регистров CardBus
!cmreslist<список ресурсов CM>	Сделать снимок памяти списка ресурсов CM
!cxr	Сделать снимок памяти записи контекста по указанному адресу
!db<физический адрес>	Отобразить физическую память в виде типа данных BYTE
!dblink<адрес>[счетчик][смещение]	Сделать снимок памяти списка при проходе по его связям с задними элементами
!dcs<Bus>.<Dev>.<Fn>	Сделать снимок памяти PCI ConfigSpace устройства
!dd<физический адрес>	Отобразить физическую память в виде типа данных DWORD
!defwrites	Сделать снимок памяти отложенной очереди на запись и устанавливает очередность кэшированных отметок записи
!devext<адрес><тип> адресу <адрес> типа <тип> <тип>	Сделать снимок памяти расширения устройства по PCI, PCMCIA, USBD, OpenHCI, USBHUB, UHCD, HID
!devnode<узел>[флаги] [системный вызов]	Сделать снимок памяти узла устройства узел: 0 Выводит главное дерево 1 Выводит отложенные удаления 2 Выводит отложенные изъятия addr Выводит указанный узел флаги: 1 Сделать снимок памяти дочерних элементов 2 Сделать снимок памяти списка ресурсов CM 4 Сделать снимок памяти списка ресурсов ввода-вывода

Команда	Описание
	8 Сделать снимок памяти преобразованного списка ресурсов CM 10 Сделать снимок памяти одних только не запущенных узлов 20 Сделать снимок памяти только узлов с проблемами Системный вызов: если задан, выводится Сделать снимок памяти только узлов, управляемых этим системным вызовом
!devobj<устройство>	Сделать снимок памяти объекта устройства и очереди IRP <устройство> имя или адрес объекта устройства
!devstack<устройство>	Сделать снимок памяти стека, сопоставленного объекту устройства
!dflink<адрес>[счетчик][смещение]	Сделать снимок памяти списка при проходе по его связям с передними элементами
смещение	Маска бит, не учитываемых в каждом указателе
!drivers	Отобразить информацию о всех загруженных системных модулях
!drvobj<драйвер>[флаги]	Сделать снимок памяти объекта драйвера и связанную с этим информацию <драйвер> Имя или адрес объекта драйвера флаги: 1 Сделать снимок памяти списка объектов устройств 2 Сделать снимок памяти точек ввода драйвера
!eb<физический адрес><список BYTE>	Ввести величины типа BYTE в физическую память
!ed<физический адрес><список DWORD>	Ввести величины типа DWORD в физическую память
!errlog	Сделать снимок памяти содержимое журнала ошибок
!exca<BasePort>.<SktNum>	Сделать снимок регистров ExCA
!exqueue[флаги]	Сделать снимок памяти ExWorkerQueues флаги: 1/2/4 то же, что !thread / !process 10 Только критичную рабочую очередь 20 Только отложенную рабочую очередь 40 Только самую критичную рабочую очередь
!exr<адрес>	Сделать снимок памяти записи исключения по указанному адресу
!filecache	Сделать снимок памяти информации о кэше файловой системы
!filelock<адрес>	Сделать снимок памяти структуры блокированного файла
!filetime	Сделать снимок памяти 64-битной структуры FILETIME в удобочитаемом виде
!fpsearch <адрес>	Находит место освобожденного специального пула
!frag[флаги]	Сделать снимок памяти фрагментации пула режима ядра флаги: 1 Выводит всю информацию о фрагментах 2 Выводит информацию о выделении памяти 3 И то и другое

продолжение

Таблица А.3 (*продолжение*)

Команда	Описание
!gentable<адрес>	Сделать снимок памяти заданной таблицы rtl_generic_table
!handle<адрес><флаги><процесс><TypeName>	Сделать снимок памяти дескриптора процесса флаги: 2 Сделать снимок памяти невыгружаемого объекта
!heap <адрес>[флаги]	Сделать снимок памяти кучи процесса адрес: Требуемая куча или 0 для всех куч флаги: -v Подробно -f Свободные элементы списка -a Все элементы -s Сводка -x Принудительный снимок, даже если данные испорчены
!help	Вывести справку по командам
!HidPpd<адрес><флаги>	Сделать снимок памяти предварительно обработанных данных устройства HID
!ib<порт>	Прочитать BYTE из порта ввода-вывода
!id<порт>	Прочитать DWORD из порта ввода-вывода
!ioapic[база]	Сделать снимок памяти APIC ввода-вывода
!ioreslist<список ресурсов IO>	Сделать снимок памяти списка требований ресурсов ввода-вывода
!irp<адрес><уровень снимка>	Сделать снимок памяти IRP по заданному адресу адрес == 0 Сделать снимок памяти активных IRP (только для установленных) уровень снимка: 0 Основная информация стека 1 Полный снимок 2 Включая информацию слежения (только для установленных)
!irpfind[тип пула][адрес перезапуска][<irpsearch><адрес>]	Найти пул для активных пакетов IRP тип пула: 0 Невыгружаемый пул (по умолчанию) 1 Выгружаемый пул 2 Специальный пул адрес перезаупуска: если указан, поиск будет перезапущен с этого места <irpsearch> Задает критерий отбора для поиска конкретного IRP: userevent Irp.UserEvent == <адрес> device Расположение стека: DeviceObject == <адрес> fileobject Irp.Tail.Overlay.OriginalFileObject == <адрес> mdlprocess Irp.MdlAddress.Process == <адрес> thread Irp.Tail.Overlay.Thread == <адрес> arg Один из аргументов == <адрес>
!iw<порт>	Прочитать WORD из порта ввода-вывода
!job<адрес>[<флаги>]	Сделать снимок памяти JobObject по адресу <адрес>, обрабатываемого в задании
!locks[-v]<адрес>	Сделать снимок памяти блокировок ресурсов режима ядра

Команда	Описание
!lookaside<адрес> <параметры> <глубина>	Сделать снимок памяти списков подстановок <адрес> параметры: 1 Сбросить счетчики списков 2 Установить глубину списка в значение <глубина>
!lpc	Сделать снимок памяти портов и сообщений LPC
!memusage	Сделать снимок памяти кадра страницы таблицы базы данных
!mps	Сделать снимок памяти структур MPS BIOS
!mtrr	Сделать снимок памяти MTTR
!npx[база]	Сделать снимок памяти области сохранения NPX
!ob<порт>	Записать BYTE в порт ввода-вывода
!obja<TypeName>	Сделать снимок памяти атрибутов объекта диспетчера объектов
!object<-r | Path | адрес | 0 TypeName>	Сделать снимок памяти объекта диспетчера объектов -r Принудительно перезагрузить кэшированные указатели на объекты
!od <порт>	Записать DWORD в порт ввода-вывода
!ow <порт>	Записать WORD в порт ввода-вывода
!patch	Включает и отключает различные флаги драйвера
!pci [флаг][шина][устройство][функция]	Сделать снимок памяти конфигурации PCI type1 [rawdump:minaddr][maxaddr] флаг: 0x01 Подробно 0x02 С шины 0 на 'шину' 0x04 Сделать снимок памяти «сырых» данных типа BYTE 0x08 Сделать снимок памяти «сырых» типа DWORD 0x10 Не пропускать некорректные устройства 0x20 Не пропускать некорректные функции 0x40 Если найдено сделать снимок памяти характеристик 0x80 Сделать снимок памяти особого устройства с VendorId:8086
!pciir	Сделать снимок памяти таблицы маршрутов PCI IRQ
!pcitree	Сделать снимок памяти структуры дерева PCI
!pcr	Сделать снимок памяти управляющей области процессора (PCR, Processor Control Region)
!pfn	Сделать снимок памяти кадра страницы записи базы данных для физической страницы
!pic	Сделать снимок памяти информации PIC (8259)
!pnpevent<адрес>	Сделать снимок памяти указанного события PNP, или всех событий, если <адрес> == 0
!pocaps	Сделать снимок памяти характеристик энергопитания системы
!podev<devobj>	Сделать снимок памяти относящихся к питанию данных в объекте устройства

продолжение

Таблица А.3 (*продолжение*)

Команда	Описание
!polist	Сделать снимок памяти списка последовательных IRP энергопитания
!polist [<devobj>]	Сделать снимок памяти элементов списка последовательных IRP энергопитания для заданного devobj
!ponode	Сделать снимок памяти стека узла устройства энергопитания (devnodes в порядке питания)
!popolicy	Сделать снимок памяти политики энергопитания системы
!poproc<Address>	Сделать снимок памяти состояния энергопотребления процессора
!pool<адрес>[детализация]	Сделать снимок памяти кучи режима ядра адрес: 0 Только куча процесса (по умолчанию) -1 Все кучи процесса иначе Элемент пула по умолчанию: 0 Итоговая информация 1 Сводка и расположение/размер областей 2 Отобразить информацию только для адреса 3 Сводка и блоки в выделенных областях (commited regions) 4 Сводка и свободные списки
!poolfind<tag>[тип пула]	Поиск вхождений заданного тега <tag> пула <tag> Четырехсимвольный тег, * и ? — групповые символы тип пула: 0 Невыгружаемый пул 1 Выгружаемый пул 2 Специальный пул
!poolused[флаги[TAG]]	Сделать снимок памяти использования по тегу пула флаги: 1 Подробно 2 Отсортировать по использованию невыгружаемого пула (NonPagedPool Usage) 4 Отсортировать по использованию выгружаемого пула (PagedPool Usage)
!poReqList[<devobj>]	Сделать снимок памяти PoRequestedPowerIrp созданных пакетов Power IRP
!portcls<devobj>[флаги]	Сделать снимок памяти данных portcls для devobj флаги: 1 Снимок порта 2 Снимок фильтра 4 Снимок pin 8 Контекст устройства 10 Информация об энергопотреблении 100 Подробно 200 Очень подробно
!potrigger<адрес>	Сделать снимок памяти значения POP_ACTION_TRIGGER

Команда	Описание
!process[флаги][имя образа]	Сделать снимок памяти процесса по указанному адресу флаги: 1 Не останавливаться после информации Cid/Image 2 Снимок состояний ожидания потока 4 Снимок только состояний ожидания потока 6 Снимок состояний ожидания и стека потока
!processfields	Показать смещения всех полей в структуре EPROCESS
!pte	Сделать снимок памяти записей PDE и PTE, соответствующих введенному адресу
!ptov<PhysicalPageNumber>	Сделать снимок памяти все корректные отображения физических страниц на виртуальные для каталога страниц
!qlocks	Сделать снимок памяти состояния всех спин-блокировок в очереди
!range<RtlRangeList>	Сделать снимок памяти значения RTL_RANGE_LIST
!ready	Сделать снимок памяти состояния всех готовых системных потоков
!reghash	Сделать снимок памяти хэш-таблицы реестра
!regkcb<адрес>	Сделать снимок памяти управляющих блоков ключей реестра
!regpool[s|r]	Сделать снимок памяти выделенного реестру страничного пула s Сохраняет список страниц реестра во временный файл r Восстанавливает список страниц реестра из временного файла
!rellist<список связей>[флаги]	Сделать снимок памяти списков связей PNP флаги: 1 Не используется 2 Снимок списка ресурсов CM 4 Снимок списка ресурсов ввода-вывода 6 Снимок преобразованного списка ресурсов CM
!remlock	Сделать снимок памяти структуры удаления блокировки
!sd<Address>[флаги]	Показать значение SECURITY_DESCRIPTOR
!sel[селектор]	Изучить значения селектора
!session<id>[флаги][имя образа]	Сделать снимок памяти сеансов
!sid <Address>[]	Отобразить структуру SID по указанному адресу
!socket<адрес>	Сделать снимок памяти структуры сокета PCMCIA
!srb<адрес>	Сделать снимок памяти SCSI Request Block (блока запросов) по указанному адресу
!stacks<степень детализации>	Сделать снимок памяти сводных данных текущих стеков ядра уровень детализации: 0 Отображает сводную информацию о стеке 1 Отображает стеки без параметров 2 Отображает стеки со всеми параметрами
!sysptes	Сделать снимок памяти системных записей PTE

продолжение

Таблица А.3 (*продолжение*)

Команда	Описание
!thread<адрес>[флаги]	Сделать снимок памяти потока по указанному адресу флаги: 1 Не используется 2 Снимок состояний ожидания потока 4 Снимок только состояний ожидания потока 6 Снимок состояний ожидания и стека потока
!threadfields	Показать смещения всех полей в структуре ETHREAD
!time	Вывести значения PerformanceCounterRate и TimerDifference
!timer	Сделать снимок памяти дерева таймера
!token<адрес>[флаги]	Сделать снимок памяти токена по указанному адресу
!tokenfields	Показать смещения всех полей в структуре токена
!trap[база]	Сделать снимок памяти кадра ловушки (trap frame)
!tss[регистр]	Сделать снимок памяти TSS
!tunnel<адрес>	Сделать снимок памяти туннельного кэша свойств файла
!tz[<адрес><флаги>]	Сделать снимок памяти тепловых зон (Без аргументов: снимок всех зон)
!tzinfo<адрес>	Сделать снимок памяти информации тепловой зоны
!urb<адрес><флаги>	Сделать снимок памяти USB Request Block (блока запросов)
!usblog<log>[addr][флаги]	Вывести журнал USB <log> USBHUB, USBD, UHCD, OpenHCI addr: Адрес, начиная с которого делать снимок из <log> флаги: r Сбрасывает журнал, начиная делать снимок с самой новой записи -sL Ищет теги в разделяемом запятыми списке L -lN Установить в N число отображаемых строк
!usbstruc<адрес><тип>	Сделать снимок памяти дескриптора (descriptor) USBHC типа <тип> <тип> OHCIReg, HCCA, OHCIHcdED, OHCIHcdTD, OHCIEndpoint, DevData, UHCDReg
!vad	Сделать снимок памяти значений VAD
!version	Версия DLL расширения
!vm	Сделать снимок памяти значений виртуальной памяти
!vpd<адрес>	Сделать снимок памяти блока параметров тома
!vtop DirBase address	Сделать снимок памяти физической страницы для виртуального адреса
!wdmaud<адрес><флаги>	Сделать снимок памяти данных wdmaud для структур флаги: 1 Снимок протокола управления вводом-выводом заданного WdmaIoctlHistoryListHead 2 Отложенные пакеты IRP заданного WdmaPendingIrpListHead 4 Выделенные MDL заданного WdmaAllocatedMdlListHead 8 Снимок pContext данного WdmaContextListHead 100 Подробно
!zombies	Найти все потерянные процессы

Таблица А.4. Прямые команды, экспортируемые userkdx.dll

Команда	Описание
!atom	Сделать снимок памяти атомов или таблиц атомов
!dcls [pcls]	Сделать снимок памяти класса окна
!dcss	Сделать снимок памяти записей стека критической секции
!dcur -avip [pcur]	Сделать снимок памяти курсоров
!dde -vr [conv|window|xact]	Сделать снимок памяти информации о слежении за DDE
!ddesk -vh <pdesk>	Показать объекты рабочего стола
!ddl [pdesk]	Сделать снимок памяти журнала рабочего стола
!ddk <pKbdTbl>	Сделать снимок памяти таблицы клавиш прерывания (deadkey table)
!df []![-p pid]	Показать или установить флаги отладки
!dfa	Сделать снимок памяти слежения за стеком при отказах в выделении памяти
!dha address	Сделать снимок памяти выделений памяти из кучи и сверить с кучей
!dhe [указатель|дескриптор] ![-t[o[p]] type [pti/ppi]]	Сделать снимок памяти записи дескрипторов
!dhk -ag [pti]	Сделать снимок памяти перехватчиков
!dhot	Сделать снимок памяти зарегистрированных горячих клавиш
!dhs -vpty [id|тип]	Сделать снимок памяти статистики таблицы дескрипторов
!di	Показать глобальные переменные обработки ввода USER
!dii <piiex>	Сделать снимок памяти расширенной информации IME
!dimc [-hrvus] -[wci] [imcl|wnd, etc.]	Сделать снимок памяти контекста ввода (Input Context)
!dimk [pImeHotKeyObj]	Сделать снимок памяти горячих клавиш IME
!dinp -v [pDeviceInfo]	Сделать снимок памяти диагностики ввода
!dkl -akv <pkl>	Сделать снимок памяти структур раскладки клавиатуры
!dll [*]addr[1#][b#][o#][c#][t[addr]]	Сделать снимок памяти связного списка (также Ctrl-C)
!dlr <указатель|дескриптор>	Показать назначенные объекту блокировки
!dm -vris <меню|окно>	Сделать снимок памяти меню
!dmon <pMonitor>	Сделать снимок памяти MONITOR
!dmq [-ac][pq]	Показать список сообщений из очередей
!dms <MenuState>	Сделать снимок памяти pMenuState
!dp -vcpt[id]	Отобразить сокращенную информацию о процессе
!dpa –cvsfrp	Сделать снимок памяти выделений памяти под пулы
!dpi [ppi]	Показать указанную структуру PROCESS_INFO
!dpm <ppopupmenu>	Сделать снимок памяти всплывающего меню
!dq -t [pq]	Сделать снимок памяти указанной структуры Q
!dsbt <pSBTrack>	Показать структуру Scroll Bar Track
!dsbwnd <psbwnd>	Сделать снимок памяти дополнительных полей окон прокрутки
!dsi [-bchmopvw]	Показать структуру SERVERINFO
!dsms -vl [psms]	Показать указанную структуру SMS (SendMessage)

продолжение

Команда	Описание
!dso <Structure>[Field][addr[*n]]	Сделать снимок памяти смещения(ий) и значения(ий) поля(ей) структуры
!dt -gvcp [id]	Отобразить сокращенную информацию о потоке
!dtdb[pdtdb]	Сделать снимок памяти Task DataBase
!dti[pti]	Показать структуру THREADINFO
!dtl[-t][указатель|дескриптор]	Показать блокировки потока
!dtmr[ptmr]	Сделать снимок памяти структуры таймера
!du[указатель|дескриптор]	Обобщенная подпрограмма создания снимка памяти объекта
!dumphmgr[-s]	Сделать снимок памяти счетчиков выделения объекта (только в отладочной версии)
!dup	Предпочтения пользователя в виде DWORD
!dupm	Битовая маска предпочтений пользователя
!dvs -s	Сделать снимок памяти разделов и отображенных в память представлений (mapped views)
!dw –aefhvsprwoz [hwnd/pwnd]	Показать информацию об окнах в системе
!dwe [-n][addr]	Показать перехватчики/уведомления (hooks/notifies) WinEvent
!dwpi -p [pwpi|ppi]	Показать заданную структуру WOWPROCESINFO
!dws [pws]	Сделать снимок памяти состояний окна
!dy [pdi]	Сделать снимок памяти DISPLAYINFO
!find baseaddr addr [o#]	Найти элемент связного списка
!fno <адрес>	Найти ближайший объект
!frr <psrcLo><psrcHi> <prefLo>[prefHi]	Найти Range Reference
!help -v [cmd]	Вывести справку по командам (-v подробно)
!hh	Сделать снимок памяти gdwHydraHint
!kbd -au [pq]	Показать ключевое состояние для очереди
!sas [-s]<addr>[длина]	Stack Analysis Stuff
!test	Протестировать основные функции отладки
!uver	Показать версии USETEXTS и WIN32K.SYS

Б Функции API ядра

В приложении Б собраны функции, экспортируемые системными модулями win32k.sys, ntdll.dll и ntoskrnl.exe00e, обсуждаемые в главе 2.

Таблица Б.1. Native API Windows 2000

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtAcceptConnectPort	0x0000			нет	нет
NtAccessCheck	0x0001			нет	нет
NtAccessCheckAndAuditAlarm	0x0002			нет	нет
NtAccessCheckByType	0x0003			нет	нет
NtAccessCheckByTypeAnd-AuditAlarm	0x0004			нет	нет
NtAccessCheckByTypeResultList	0x0005			нет	нет
NtAccessCheckByTypeResult-ListAndAuditAlarm	0x0006			нет	нет
NtAccessCheckByTypeResult-ListAndAuditAlarmByHandle	0x0007			нет	нет
NtAddAtom	0x0008			нет	нет
NtAdjustGroupsToken	0x0009			нет	нет
NtAdjustPrivilegesToken	0x000A			нет	нет
NtAlertResumeThread	0x000B			нет	нет
NtAlertThread	0x000C			нет	нет
NtAllocateLocallyUniqueId	0x000D			нет	нет
NtAllocateUserPhysicalPages	0x000E			нет	нет
NtAllocateUuids	0x000F				нет
NtAllocateVirtualMemory	0x0010				
NtAreMappedFilesTheSame	0x0011			нет	нет
NtAssignProcessToJobObject	0x0012			нет	нет
NtBuildNumber	нет	нет	нет		нет
NtCallbackReturn	0x0013			нет	нет

продолжение

Таблица Б.1 (*продолжение*)

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtCancelDeviceWakeupRequest	0x0016			нет	нет
NtCancelIoFile	0x0014			нет	
NtCancelTimer	0x0015			нет	
NtClearEvent	0x0017			нет	
NtClose	0x0018				
NtCloseObjectAuditAlarm	0x0019			нет	
NtCompleteConnectPort	0x001A			нет	нет
NtConnectPort	0x001B				
NtContinue	0x001C			нет	нет
NtCreateChannel	0x00F1			нет	нет
NtCreateDirectoryObject	0x001D			нет	
NtCreateEvent	0x001E				
NtCreateEventPair	0x001F			нет	нет
NtCreateFile	0x0020				
NtCreateIoCompletion	0x0021			нет	нет
NtCreateJobObject	0x0022			нет	нет
NtCreateKey	0x0023			нет	
NtCreateMailslotFile	0x0024			нет	нет
NtCreateMutant	0x0025			нет	нет
NtCreateNamedPipeFile	0x0026			нет	нет
NtCreatePagingFile	0x0027			нет	нет
NtCreatePort	0x0028			нет	нет
NtCreateProcess	0x0029			нет	нет
NtCreateProfile	0x002A			нет	нет
NtCreateSection	0x002B				
NtCreateSemaphore	0x002C			нет	нет
NtCreateSymbolicLinkObject	0x002D			нет	
NtCreateThread	0x002E			нет	нет
NtCreateTimer	0x002F			нет	
NtCreateToken	0x0030			нет	нет
NtCreateWaitablePort	0x0031			нет	нет
NtCurrentTeb	нет		нет	нет	нет
NtDelayExecution	0x0032			нет	нет
NtDeleteAtom	0x0033				нет
NtDeleteFile	0x0034				
NtDeleteKey	0x0035			нет	
NtDeleteObjectAuditAlarm	0x0036			нет	нет
NtDeleteValueKey	0x0037			нет	
NtDeviceIoControlFile	0x0038				
NtDisplayString	0x0039			нет	

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtDuplicateObject	0x003A				
NtDuplicateToken	0x003B				
NtEnumerateKey	0x003C			нет	
NtEnumerateValueKey	0x003D			нет	
NtEx00tendSection	0x003E			нет	нет
NtFilterToken	0x003F			нет	нет
NtFindAtom	0x0040				нет
NtFlushBuffersFile	0x0041			нет	нет
NtFlushInstructionCache	0x0042			нет	
NtFlushKey	0x0043			нет	
NtFlushVirtualMemory	0x0044			нет	
NtFlushWriteBuffer	0x0045			нет	нет
NtFreeUserPhysicalPages	0x0046			нет	нет
NtFreVirtualMemory	0x0047				
NtFsControlFile	0x0048				
NtGetContex00tThread	0x0049			нет	нет
NtGetDevicePowerState	0x004A			нет	нет
NtGetPlugPlayEvent	0x004B			нет	нет
NtGetTickCount	0x004C			нет	нет
NtGetWriteWatch	0x004D			нет	нет
NtGlobalFlag	нет	нет	нет		нет
NtImpersonate-AnonymousToken	0x004E			нет	нет
NtImpersonateClientOfPort	0x004F			нет	нет
NtImpersonateThread	0x0050			нет	нет
NtInitializeRegistry	0x0051			нет	нет
NtInitializePowerAction	0x0052			нет	
NtIsSystemResumeAutomatic	0x0053			нет	нет
NtListenChannel	0x00F2			нет	нет
NtListenPort	0x0054			нет	нет
NtLoadDriver	0x0055			нет	
NtLoadKey	0x0056			нет	
NtLoadKey2	0x0057			нет	нет
NtLockFile	0x0058				нет
NtLockVirtualMemory	0x0059			нет	нет
NtMakeTemporaryObject	0x005A			нет	
NtMapUserPhysicalPages	0x005B			нет	нет
NtMapUserPhysicalPagesScatter	0x005C			нет	нет
NtMapViewOfSection	0x005D				
NtNotifyChangeDirectoryFile	0x005E				нет
NtNotifyChangeKey	0x005F			нет	

продолжение

Таблица Б.1 (*продолжение*)

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtNotifyChangeMultipleKey	0x0060			нет	нет
NtOpenChannel	0x00F3			нет	нет
NtOpenDirectoryObject	0x0061			нет	
NtOpenEvent	0x0062			нет	
NtOpenEventPair	0x0063			нет	нет
NtOpenFile	0x0064				
NtOpenIoCompletion	0x0065			нет	нет
NtOpenJobObject	0x0066			нет	нет
NtOpenKey	0x0067			нет	
NtOpenMutant	0x0068			нет	нет
NtOpenObjectAuditAlarm	0x0069			нет	нет
NtOpenProcess	0x006A				
NtOpenProcessToken	0x006B				
NtOpenSection	0x006C			нет	
NtOpenSemaphore	0x006D			нет	нет
NtOpenSymbolicLinkObject	0x006E			нет	
NtOpenThread	0x006F			нет	
NtOpenThreadToken	0x0070			нет	
NtOpenTimer	0x0071			нет	
NtPlugPlayControl	0x0072			нет	нет
NtPowerInformation	0x0073			нет	
NtPrivilegeCheck	0x0074			нет	нет
NtPrivilegedServiceAuditAlarm	0x0075			нет	нет
NtPrivilegeObjectAuditAlarm	0x0076			нет	нет
NtProtectVirtualMemory	0x0077			нет	нет
NtPulseEvent	0x0078			нет	
NtQueryAttributesFile	0x007A			нет	нет
NtQueryDefaultLocale	0x007B			нет	
NtQueryDefaultUILanguage	0x007C			нет	
NtQueryDirectoryFile	0x007D				
NtQueryDirectoryObject	0x007E			нет	
NtQueryEaFile	0x007F				
NtQueryEvent	0x0080			нет	нет
NtQueryFullAtributesFile	0x0081			нет	нет
NtQueryInformationAtom	0x0079				нет
NtQueryInformationFile	0x0082				
NtQueryInformationJobObject	0x0083			нет	нет
NtQueryInformationPort	0x0085			нет	нет
NtQueryInformationProcess	0x0086				
NtQueryInformationThread	0x0087			нет	нет
NtQueryInformationToken	0x0088				

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtQueryInstallUILanguage	0x0089			нет	
NtQueryIntervalProfile	0x008A			нет	нет
NtQueryIoCompletion	0x0084			нет	нет
NtQueryKey	0x008B			нет	
NtQueryMultipleValueKey	0x008C			нет	нет
NtQueryMutant	0x008D			нет	нет
NtQueryObject	0x008E			нет	
NtQueryOpenSubKeys	0x008F			нет	нет
NtQueryPerformanceCounter	0x0090			нет	нет
NtQueryQuotaInformationFile	0x0091				нет
NtQuerySection	0x0092			нет	
NtQuerySecurityObject	0x0093				
NtQuerySemaphore	0x0094			нет	нет
NtQuerySymbolicLinkObject	0x0095			нет	
NtQuerySystem-EnvironmentValue	0x0096			нет	нет
NtQuerySystemInformation	0x0097				
NtQuerySystemTime	0x0098			нет	нет
NtQueryTimer	0x0099			нет	нет
NtQueryTimerResolution	0x009A			нет	нет
NtQueryValueKey	0x009B			нет	
NtQueryVirtualMemory	0x009C			нет	нет
NtQueryVolumeInformationFile	0x009D				
NtQueueApcThread	0x009E			нет	нет
NtRaiseEx00ception	0x009F			нет	нет
NtRaiseHardError	0x00A0			нет	нет
NtReadFile	0x00A1				
NtReadFileScatter	0x00A2			нет	нет
NtReadRequestData	0x00A3			нет	нет
NtReadVirtualMemory	0x00A4			нет	нет
NtRegisterThreadTerminatePort	0x00A5			нет	нет
NtReleaseMutant	0x00A6			нет	нет
NtReleaseSemaphore	0x00A7			нет	нет
NtRemoveIoCompletion	0x00A8			нет	нет
NtReplaceKey	0x00A9			нет	
NtReplyPort	0x00AA			нет	нет
NtReplyWaitReceivePort	0x00AB			нет	нет
NtReplyWaitReceivePortEx00	0x00AC			нет	нет
NtReplyWaitReplyPort	0x00AD			нет	нет
NtReplyWaitSendChannel	0x00F4			нет	нет
NtRequestDeviceWakeup	0x00AE			нет	нет
NtRequestPort	0x00AF				нет

продолжение

Таблица Б.1 (*продолжение*)

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtRequestWaitReplyPort	0x00B0				
NtRequestWakeupLatency	0x00B1			нет	нет
NtResetEvent	0x00B2			нет	
NtResetWriteWatch	0x00B3			нет	нет
NtRestoreKey	0x00B4			нет	
NtResumeThread	0x00B5			нет	нет
NtSaveKey	0x00B6			нет	
NtSaveMergedKeys	0x00B7			нет	нет
NtSecureConnectPort	0x00B8			нет	нет
NtSendWaitReplyChannel	0x00F5			нет	нет
NtSetContextChannel	0x00F6			нет	нет
NtSetIoCompletion	0x00B9			нет	нет
NtSetContextThread	0x00BA			нет	нет
NtSetDefaultHardErrorPort	0x00BB			нет	нет
NtSetDefaultLocale	0x00BC			нет	
NtSetDefaultUILanguage	0x00BD			нет	
NtSetEaFile	0x00BE				
NtSetEvent	0x00BF				
NtSetHighEventPair	0x00C0			нет	нет
NtSetHighWaitLowEventPair	0x00C1			нет	нет
NtSetInformationFile	0x00C2				
NtSetInformationJobObject	0x00C3			нет	нет
NtSetInformationKey	0x00C4			нет	нет
NtSetInformationObject	0x00C5			нет	
NtSetInformationProcess	0x00C6				
NtSetInformationThread	0x00C7				
NtSetInformationToken	0x00C8			нет	нет
NtSetIntervalProfile	0x00C9			нет	нет
NtSetLdtEntries	0x00CA			нет	нет
NtSetLowEventPair	0x00CB			нет	нет
NtSetLowWaitHighEventPair	0x00CC			нет	нет
NtSetQuotaInformationFile	0x00CD				нет
NtSetSecurityObject	0x00CE				
NtSetSystemEnvironmentValue	0x00CF			нет	нет
NtSetSystemInformation	0x00D0			нет	
NtSetSystemPowerState	0x00D1			нет	нет
NtSetSystemTime	0x00D2			нет	
NtSetThreadEx00ecutionState	0x00D3			нет	нет
NtSetTimer	0x00D4			нет	
NtSetTimerResolution	0x00D5			нет	нет
NtSetUuidSeed	0x00D6			нет	нет
NtSetValueKey	0x00D7			нет	

Имя функции	INT 2Eh	ntdll.Nt*	ntdll.Zw*	ntoskrnl.Nt*	ntoskrnl.Zw*
NtSetVolumeInformationFile	0x00D8				
NtShutdownSystem	0x00D9			нет	нет
NtSignalAndWaitFor-SingleObject	0x00DA			нет	нет
NtStartProfile	0x00DB			нет	нет
NtStopProfile	0x00DC			нет	нет
NtSuspendThread	0x00DD			нет	нет
NtSystemDebugControl	0x00DE			нет	нет
NtTerminateJobObject	0x00DF			нет	нет
NtTerminateProcess	0x00E0			нет	
NtTerminateThread	0x00E1			нет	нет
NtTestAlert	0x00E2			нет	нет
NtUnloadDriver	0x00E3			нет	
NtUnloadKey	0x00E4			нет	
NtUnlockFile	0x00E5				нет
NtUnlockVirtualMemory	0x00E6			нет	нет
NtUnmapViewOfSection	0x00E7			нет	
NtVdmControl	0x00E8				нет
NtWaitForMultipleObjects	0x00E9			нет	
NtWaitForSingleObject	0x00EA				
NtWaitHighEventPair	0x00EB			нет	нет
NtWaitLowEventPair	0x00EC			нет	нет
NtWriteFile	0x00ED				
NtWriteFileGather	0x00EE			нет	нет
NtWriteRequestData	0x00EF			нет	нет
NtWriteVirtualMemory	0x00F0			нет	нет
NtYieldExecution	0x00F7			нет	

Таблица Б.2. *Интерфейс GDI/Win32K*

gdi32.dll	INT 2Eh	win32k.sys
нет	0x1000	NtGdiAbortDoc
нет	0x1001	NtGdiAbortPath
нет	0x1002	NtGdiAddFontResourceW
нет	0x1003	NtGdiAddRemoteFontToDC
нет	0x1004	NtGdiAddFontMemResourceEx
нет	0x1005	NtGdiRemoveMergeFont
нет	0x1006	NtGdiAddRemoveMMInstanceToDC
нет	0x1007	NtGdiAlphaBlend
нет	0x1008	NtGdiAngleArc
AnyLinkedFonts	0x1009	NtGdiAnyLinkedFonts
FontsIsLinked	0x100A	NtGdiFontsIsLinked
нет	0x100B	NtGdiArcInternal

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
нет	0x100C	NtGdiBeginPath
нет	0x100D	NtGdiBitBlt
нет	0x100E	NtGdiCancelDC
нет	0x100F	NtGdiCheckBitmapBits
нет	0x1010	NtGdiCloseFigure
нет	0x1011	NtGdiColorCorrectPalette
нет	0x1012	NtGdiCombineRgn
нет	0x1013	NtGdiCombineTransform
нет	0x1014	NtGdiComputeXformCoefficients
GdiConsoleTextOut	0x1015	NtGdiConsoleTextOut
нет	0x1016	NtGdiConvertMetafileRect
нет	0x1017	NtGdiCreateBitmap
нет	0x1018	NtGdiCreateClientObj
нет	0x1019	NtGdiCreateColorSpace
нет	0x101A	NtGdiCreateColorTransform
нет	0x101B	NtGdiCreateCompatibleBitmap
нет	0x101C	NtGdiCreateCompatibleDC
нет	0x101D	NtGdiCreateDIBBrush
нет	0x101E	NtGdiCreateDIBitmapInternal
нет	0x101F	NtGdiCreateDIBSection
нет	0x1020	NtGdiCreateEllipticRgn
CreateHalftonePalette	0x1021	NtGdiCreateHalftonePalette
нет	0x1022	NtGdiCreateHatchBrushInternal
нет	0x1023	NtGdiCreateMetafileDC
нет	0x1024	NtGdiCreatePaletteInternal
нет	0x1025	NtGdiCreatePatternBrushInternal
нет	0x1026	NtGdiCreatePen
нет	0x1027	NtGdiCreateRectRgn
нет	0x1028	NtGdiCreateRoundRectRgn
нет	0x1029	NtGdiCreateServerMetaFile
нет	0x102A	NtGdiCreateSolidBrash
нет	0x102B	NtGdiD3dContextCreate
нет	0x102C	NtGdiD3dContextDestroy
нет	0x102D	NtGdiD3dContextDestroyAll
нет	0x102E	NtGdiD3dValidateTextureStageState
нет	0x102F	NtGdiD3dDrawPrimitives2
нет	0x1030	NtGdiDdGetDriverState
нет	0x1031	NtGdiDdAddAttachedSurface
нет	0x1032	NtGdiDdAlphaBlt
нет	0x1033	NtGdiDdAttachSurface
нет	0x1034	NtGdiDdBeginMoCompFrame
нет	0x1035	NtGdiDdBlt

gdi32.dll	INT 2Eh	win32k.sys
нет	0x1036	NtGdiDdCanCreateSurface
нет	0x1037	NtGdiDdCanCreateD3DBuffer
нет	0x1038	NtGdiDdColorControl
нет	0x1039	NtGdiDdCreateDirectDrawObject
нет	0x103A	NtGdiDdCreateSurface
нет	0x103B	NtGdiDdCreateSurface[1]
нет	0x103C	NtGdiDdCreateMoComp
нет	0x103D	NtGdiDdCreateSurfaceObject
нет	0x103E	NtGdiDdDeleteDirectDrawObject
нет	0x103F	NtGdiDdDeleteSurfaceObject
нет	0x1040	NtGdiDdDestroyMoComp
нет	0x1041	NtGdiDdDestroySurface
нет	0x1042	NtGdiDdDestroyD3DBuffer
нет	0x1043	NtGdiDdEndMoCompFrame
нет	0x1044	NtGdiDdFlip
нет	0x1045	NtGdiDdFlipToGDISurface
нет	0x1046	NtGdiDdGetAvailDriverMemory
нет	0x1047	NtGdiDdGetBltStatus
нет	0x1048	NtGdiDdGetDC
нет	0x1049	NtGdiDdGetDriverInfo
нет	0x104A	NtGdiDdGetDxHandle
нет	0x104B	NtGdiDdGetFlikpStatus
нет	0x104C	NtGdiDdGetInternalMoCompInfo
нет	0x104D	NtGdiDdGetMoCompBuffInfo
нет	0x104E	NtGdiDdGetMoCompGuids
нет	0x104F	NtGdiDdGetMoCompFormats
нет	0x1050	NtGdiDdGetScanLine
нет	0x1051	NtGdiDdLock
нет	0x1052	NtGdiDdLockD3D
нет	0x1053	NtGdiDdQueryDirectDrawObject
нет	0x1054	NtGdiDdQueryMoCompStatus
нет	0x1055	NtGdiDdReenableDirectDrawObject
нет	0x1056	NtGdiDdReleaseDC
нет	0x1057	NtGdiDdRenderMoComp
нет	0x1058	NtGdiDdResetVisrgn
нет	0x1059	NtGdiDdSetColorKey
нет	0x105A	NtGdiDdSetExclusiveMode
нет	0x105B	NtGdiDdSetGammaRamp

продолжение

[1] Данное имя здесь и в оригинальной книге повторяет предыдущее. К сожалению, мне не удалось проверить правильность этой таблицы. Я буду благодарен читателям за любую информацию на эту тему. — *Примеч. ред.*

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
нет	0x105C	NtGdiDdCreateSurfaceEx
нет	0x105D	NtGdiDdSetOverlayPosition
нет	0x105E	NtGdiDdUnattachSurface
нет	0x105F	NtGdiDdUnlock
нет	0x1060	NtGdiDdUnlockD3D
нет	0x1061	NtGdiDdUpdateOverlay
нет	0x1062	NtGdiDdWaitForVerticalBlank
нет	0x1063	NtGdiDvpCanCreateVideoPort
нет	0x1064	NtGdiDvpColorControl
нет	0x1065	NtGdiDvpCreateVideoPort
нет	0x1066	NtGdiDvpDestroyVideoPort
нет	0x1067	NtGdiDvpFlipVideoPort
нет	0x1068	NtGdiDvpGetVideoPortBandwith
нет	0x1069	NtGdiDvpGetVideoPortField
нет	0x106A	NtGdiDvpGetVideoPortFlipStatus
нет	0x106B	NtGdiDvpGetVideoPortInputFormats
нет	0x106C	NtGdiDvpGetVideoPortLine
нет	0x106D	NtGdiDvpGetVideoPortOutputFormats
нет	0x106E	NtGdiDvpGetVideoPortConnectInfo
нет	0x106F	NtGdiDvpGetVideoSignalStatus
нет	0x1070	NtGdiDvpUpdateVideoPort
нет	0x1071	NtGdiDvpWaitForVideoPortSync
нет	0x1072	NtGdiDeleteClientObj
нет	0x1073	NtGdiDeleteColorSpace
нет	0x1074	NtGdiDeleteColorTransform
нет	0x1075	NtGdiDeleteObjectApp
нет	0x1076	NtGdiDescribePixelFormat
нет	0x1077	NtGdiGetPerBandInfo
нет	0x1078	NtGdiDoBanding
нет	0x1079	NtGdiDoPalette
нет	0x107A	NtGdiDrawEscape
нет	0x107B	NtGdiEllipse
EnableEUDC	0x107C	NtGdiEnableEudc
нет	0x107D	NtGdiEndDoc
нет	0x107E	NtGdiEndPage
нет	0x107F	NtGdiEndPath
нет	0x1080	NtGdiEnumFontChunk
нет	0x1081	NtGdiEnumFontClose
нет	0x1082	NtGdiEnumFontOpen
нет	0x1083	NtGdiEnumObjects
нет	0x1084	NtGdiEqualRgn

gdi32.dll	INT 2Eh	win32k.sys
нет	0x1085	NtGdiEudcEnumFaceNameLinkW
нет	0x1086	NtGdiEudcLoadUnloadLink
нет	0x1087	NtGdiExcludeClipRect
нет	0x1088	NtGdiExtCreatePen
нет	0x1089	NtGdiExtCreatgeRegion
нет	0x108A	NtGdiExtEscape
нет	0x108B	NtGdiExtFloodFill
нет	0x108C	NtGdiExtGetObjectW
нет	0x108D	NtGdiExtSelectClipRgn
нет	0x108E	NtGdiExtTextOutW
нет	0x108F	NtGdiFillPath
нет	0x1090	NtGdiFillRgn
нет	0x1091	NtGdiFlattenPath
нет	0x1092	NtGdiFlushUserBatch
нет	0x1093	GreFlush
нет	0x1094	NtGdiForceUFIMapping
нет	0x1095	NtGdiFrameRgn
нет	0x1096	NtGdiFullScreenControl
нет	0x1097	NtGdiGetAndSetDCDword
нет	0x1098	NtGdiGetAppClipBox
нет	0x1099	NtGdiGetBitmapBits
нет	0x109A	NtGdiGetBitmapDimension
нет	0x109B	NtGdiGetBoundsRect
нет	0x109C	NtGdiGetCharABCWidthsW
нет	0x109D	NtGdiGetCharacterPlacementW
нет	0x109E	NtGdiGetCharSet
нет	0x109F	NtGdiGetCharWidthW
GetCharWidthInfo	0x10A0	NtGdiGetCharWidthInfo
нет	0x10A1	NtGdiGetColorAdjustment
нет	0x10A2	NtGdiGetColorSpaceForBitmap
нет	0x10A3	NtGdiGetDCDword
нет	0x10A4	NtGdiGetDCforBitmap
нет	0x10A5	NtGdiGetDCObject
нет	0x10A6	NtGdiGetDCPoint
нет	0x10A7	NtGdiGetDeviceCaps
нет	0x10A8	NtGdiGetDeviceGammaRamp
нет	0x10A9	NtGdiGetDeviceCapsAll
нет	0x10AA	NtGdiGetDIBitsInternal
нет	0x10AB	NtGdiGetETM
нет	0x10AC	NtGdiGetEudcTimeStampEx
нет	0x10AD	NtGdiGetFontData

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
нет	0x10AE	NtGdiGetFontResourceInfoInternalW
GetGliphIndIciesW	0x10AF	NtGdiGetGliphIndiciesW
нет	0x10B0	NtGdiGetGliphIndiciesWInternal
нет	0x10B1	NtGdiGetGlyphOutline
нет	0x10B2	NtGdiGetKerningPairs
нет	0x10B3	NtGdiGetLinkedUFIs
нет	0x10B4	NtGdiGetMiterLimit
нет	0x10B5	NtGdiGetMonitorID
нет	0x10B6	NtGdiGetNearestColor
нет	0x10B7	NtGdiGetNearestPaletteIndex
нет	0x10B8	NtGdiGetObjectBitmapHandle
нет	0x10B9	NtGdiGetOutlineTextMetricsInternalW
нет	0x10BA	NtGdiGetPath
нет	0x10BB	NtGdiGetPixel
нет	0x10BC	NtGdiGetRandomRgn
нет	0x10BD	NtGdiGetRasterizerCaps
нет	0x10BE	NtGdiGetRealizationInfo
нет	0x10BF	NtGdiGetRegionData
нет	0x10C0	NtGdiGetRgnBox
нет	0x10C1	NtGdiGetServerMetaFileBits
GdiGetSpoolMessage	0x10C2	NtGdiGetSpoolMessage
нет	0x10C3	NtGdiGetStats
нет	0x10C4	NtGdiGetStockObject
нет	0x10C5	NtGdiGetStringBitmapW
нет	0x10C6	NtGdiGetSystemPaletteUse
GetTextCharsetInfo	0x10C7	NtGdiGetTextCharsetInfo
нет	0x10C8	NtGdiGetTextExtent
нет	0x10C9	NtGdiGetTextExtentExW
нет	0x10CA	NtGdiGetTextFaceW
нет	0x10CB	NtGdiGetTextMetricsW
нет	0x10CC	NtGdiGetTransform
нет	0x10CD	NtGdiGetUFI
нет	0x10CE	NtGdiGetUFIPathname
GetFontUnicodeRanges	0x10CF	NtGdiGetFontUnicodeRanges
нет	0x10D0	NtGdiGetWidthTable
нет	0x10D1	NtGdiGradientFill
нет	0x10D2	NtGdiHfontCreate
нет	0x10D3	NtGdiIcmBrushInfo
нет	0x10D4	NtGdiInit
GdiInitSpool	0x10D5	NtGdiInitSpool
нет	0x10D6	NtGdiIntersectClipRect

gdi32.dll	INT 2Eh	win32k.sys
нет	0x10D7	NtGdiInvertRgn
нет	0x10D8	NtGdi LineTo
нет	0x10D9	NtGdiMakeFontDir
нет	0x10DA	NtGdiMakeInfoDC
нет	0x10DB	NtGdiMaskBlt
нет	0x10DC	NtGdiModifyWordTransform
нет	0x10DD	NtGdiMonoBitmap
нет	0x10DE	NtGdIMoveTo
нет	0x10DF	NtGdiOffsetClipRgn
нет	0x10E0	NtGdiOffsetRgn
нет	0x10E1	NtGdiOpenDCW
нет	0x10E2	NtGdiPatBlt
нет	0x10E3	NtGdiPolyPatBlt
нет	0x10E4	NtGdiPathToRegion
нет	0x10E5	NtGdiPlgBlt
нет	0x10E6	NtGdiPolyDraw
нет	0x10E7	NtGdiPolyPolyDraw
нет	0x10E8	NtGdiPolyTextOutW
нет	0x10E9	NtGdiPtInRegion
нет	0x10EA	NtGdiPtVisible
GdiQueryFonts	0x10EB	NtGdiQueryFonts
нет	0x10EC	NtGdiQueryFontAssocInfo
нет	0x10ED	NtGdiRectangle
нет	0x10EE	NtGdiRectInRegion
нет	0x10EF	NtGdiRectVisible
нет	0x10F0	NtGdiRemoveFontResourceW
нет	0x10F1	NtGdiRemoveFontMemResourceEx
нет	0x10F2	NtGdiResetDC
нет	0x10F3	NtGdiResizePalette
нет	0x10F4	NtGdiRestoreDC
нет	0x10F5	NtGdiRoundRect
нет	0x10F6	NtGdiSaveDC
нет	0x10F7	NtGdiScaleViewportExtEx
нет	0x10F8	NtGdiScaleWindowExtEx
нет	0x10F9	NtGdiSelectBitmap
нет	0x10FA	NtGdiSelectBrush
нет	0x10FB	NtGdiSelectClipPath
нет	0x10FC	NtGdiSelectFont
нет	0x10FD	NtGdiSelectPen
нет	0x10FE	NtGdiSetBitmapBits
нет	0x10FF	NtGdiSetBitmapDimension

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
нет	0x1100	NtGdiSetBoundsRect
нет	0x1101	NtGdiSetBrushOrg
нет	0x1102	NtGdiSetColorAdjustment
нет	0x1103	NtGdiSetColorSpace
нет	0x1104	NtGdiSetDeviceGammaRamp
нет	0x1105	NtGdiSetDIBitsToDeviceInternal
нет	0x1106	NtGdiSetFontEnumeration
нет	0x1107	NtGdiSetFontXform
нет	0x1108	NtGdiSetIcmMode
нет	0x1109	NtGdiSetLinkedUFIs
SetMagicColors	0x110A	NtGdiSetMagicColors
нет	0x110B	NtGdiSetMetaRgn
нет	0x110C	NtGdiSetMiterLimit
нет	0x110D	NtGdiGetDeviceWidth
нет	0x110E	NtGdiMirrorWindowOrg
нет	0x110F	NtGdiSetLayout
нет	0x1110	NtGdiSetPixel
нет	0x1111	NtGdiSetPixelFormat
нет	0x1112	NtGdiSetRectRgn
нет	0x1113	NtGdiSetSystemPaletteUse
нет	0x1114	NtGdiSetTextJustification
нет	0x1115	NtGdiSetupPublicCFONT
нет	0x1116	NtGdiSetVirtualResolution
нет	0x1117	NtGdiSetSizeDevice
нет	0x1118	NtGdiStartDoc
нет	0x1119	NtGdiStartPage
нет	0x111A	NtGdiStretchBlt
нет	0x111B	NtGdiStretchDIBitsInternal
нет	0x111C	NtGdiStrokeAndFillPath
нет	0x111D	NtGdiStrokePath
нет	0x111E	NtGdiSwapBuffers
нет	0x111F	NtGdiTransformPoints
нет	0x1120	NtGdiTransparentBit
нет	0x1121	NtGdiUnloadPrinterdriver
нет	0x1122	NtGdiUnmapMemFont
нет	0x1123	NtGdiUnrealizeObject
нет	0x1124	NtGdiUpdateColors
нет	0x1125	NtGdiWidenPath
ActivateKeyboardLayput	0x1126	NtUserActivateKeyboardLayput
нет	0x1127	NtUserAlterWindowStyle
нет	0x1128	NtUserAssociateInputContext

gdi32.dll	INT 2Eh	win32k.sys
AttachThreadInput	0x1129	NtUserAttachThreadInput
BeginPaint	0x112A	NtUserBeginPaint
нет	0x112B	NtUserBitBltSysBmp
BlockInput	0x112C	NtUserBlockInput
нет	0x112D	NtUserBuildHimcList
нет	0x112E	NtUserBuildHwndList
нет	0x112F	NtUserBuildNameList
нет	0x1130	NtUserBuildPropList
нет	0x1131	NtUserCallHwnd
нет	0x1132	NtUserCallHwndLock
нет	0x1133	NtUserCallHwndOpt
нет	0x1134	NtUserCallHwndParam
нет	0x1135	NtUserCallHwndParamLock
нет	0x1136	NtUserCallMsgFilter
нет	0x1137	NtUserCallNextHookEx
нет	0x1138	NtUserCallNoParam
нет	0x1139	NtUserCallOneParam
нет	0x113A	NtUserCallTwoParam
ChangeClipbpardChain	0x113B	NtUserChangeClipbpardChain
нет	0x113C	NtUserChangeDisplaySettings
нет	0x113D	NtUserCheckImeHotkey
нет	0x113E	NtUserCheckMenuItem
ChildWindowFromPointEx	0x113F	NtUserChildWindowFromPointEx
ClipCursor	0x1140	NtUserClipCursor
CloseClipboard	0x1141	NtUserCloseClipboard
CloseDesktop	0x1142	NtUserCloseDesktop
CloseWindowStation	0x1143	NtUserCloseWindowStation
нет	0x1144	NtUserConsoleControl
нет	0x1145	NtUserConvertMemHandle
CopyAcceleratorTableW	0x1146	NtUserCopyAcceleratorTableW
CountClipboardFormats	0x1147	NtUserCountClipboardFormats
CreateAcceleratorTableW	0x1148	NtUserCreateAcceleratorTableW
CreateCaret	0x1149	NtUserCreateCaret
нет	0x114A	NtUserCreateDesktop
нет	0x114B	NtUserCreateInputContext
нет	0x114C	NtUserCreateLocalMemHandle
нет	0x114D	NtUserCreateWindowEx
нет	0x114E	NtUserCreateWindowStation
DdeGetQualityOfService	0x114F	NtUserDdeGetQualityOfService
нет	0x1150	NtUserDdeInitialize
DdeSetQualityOfService	0x1151	NtUserDdeSetQualityOfService

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
DeferWindowPos	0x1152	NtUserDeferWindowPos
нет	0x1153	NtUserDefSetText
DeleteMenu	0x1154	NtUserDeleteMenu
нет	0x1155	NtUserDestroyAcceleratorTable
нет	0x1156	NtUserDestroyCursor
нет	0x1157	NtUserDestroyInputContext
DestroyMenu	0x1158	NtUserDestroyMenu
DestroyWindow	0x1159	NtUserDestroyWindow
нет	0x115A	NtUserDisableThreadIme
нет	0x115B	NtUserDispatchMessage
DractDetect	0x115C	NtUserDractDetect
DragObject	0x115D	NtUserDragObject
DrawAnimatedRects	0x115E	NtUserDrawAnimatedRects
нет	0x115F	NtUserDrawCaption
нет	0x1160	NtUserDrawCaptionTemp
нет	0x1161	NtUserDrawIconEx
нет	0x1162	NtUserDrawMenuBarTemp
EmptyClipboard	0x1163	NtUserEmptyClipboard
нет	0x1164	NtUserEnableMenuItem
EnableScrollBar	0x1165	NtUserEnableScrollBar
нет	0x1166	NtUserEndDeferWindowPosEx
EndMenu	0x1167	NtUserEndMenu
EndPaint	0x1168	NtUserEndPaint
нет	0x1169	NtUserEnumDisplayDevices
EnumDisplayMonitors	0x116A	NtUserEnumDisplayMonitors
нет	0x116B	NtUserEnumDisplaySettings
нет	0x116C	NtUserUserEvent
ExcludeUpdateRgn	0x116D	NtUserExcludeUpdateRgn
нет	0x116E	NtUserFillWindow
нет	0x116F	NtUserFindExistingCursorIcon
нет	0x1170	NtUserFindWindowEx
FlashWindowEx	0x1171	NtUserFlashWindowEx
нет	0x1172	NtUserGetAltTabInfo
GetAncestor	0x1173	NtUserGetAncestor
нет	0x1174	NtUserGetAppImeLevel
нет	0x1175	NtUserGetAsyncKeyState
GetCaretBlinkTime	0x1176	NtUserGetGetCaretBlinkTime
GetCaretPos	0x1177	NtUserGetCaretPos
нет	0x1178	NtUserGetClassInfo
нет	0x1179	NtUserGetClassName
нет	0x117A	NtUserGetClipboardData

gdi32.dll	INT 2Eh	win32k.sys
нет	0x117B	NtUserGetClipboardFormatName
GetClipboardOwner	0x117C	NtUserGetClipboardOwner
GetClipboardSequenceNumber	0x117D	NtUserGetClipboardSequenceNumber
GetClipboardViewer	0x117E	NtUserGetClipboardViewer
GetClipCursor	0x117F	NtUserGetClipCursor
GetComboBoxInfo	0x1180	NtUserGetComboBoxInfo
нет	0x1181	NtUserGetControlBrush
нет	0x1182	NtUserGetControlColor
нет	0x1183	NtUserGetCPD
нет	0x1184	NtUserGetCursorFrameInfo
GetCursorInfo	0x1185	NtUserGetCursorInfo
GetDC	0x1186	NtUserGetDC
GetDCEx	0x1187	NtUserGetDCEx
GetDoubleClickTime	0x1188	NtUserGetDoubleClickTime
GetForegroundWindow	0x1189	NtUserGetForegroundWindow
GetGuiResources	0x118A	NtUserGetGuiResources
GetGUIThreadInfo	0x118B	NtUserGetGUIThreadInfo
нет	0x118C	NtUserGetIconInfo
нет	0x118D	NtUserGetIconSize
нет	0x118E	NtUserGetImeHotKey
нет	0x118F	NtUserGetImeInfoEx
GetInternalWindowPos	0x1190	NtUserGetInternalWindowPos
GetKeyboardLayoutList	0x1191	NtUserGetKeyboardLayoutList
нет	0x1192	NtUserGetKeyboardLayoutName
GetKeyboardState	0x1193	NtUserGetKeyboardState
нет	0x1194	NtUserGetKeyNameState
нет	0x1195	NtUserGetKeyState
GetListBoxInfo	0x1196	NtUserGetListBoxInfo
GetMenuBarInfo	0x1197	NtUserGetMenuBarInfo
нет	0x1198	NtUserGetMenuIndex
GetMenuItemRect	0x1199	NtUserGetMenuItemRect
нет	0x119A	NtUserGetMessage
GetMouseMovePointsEx	0x119B	NtUserGetMouseMovePointsEx
GetObjectInformation	0x119C	NtUserGetObjectInformation
GetOpenClipboardWindow	0x119D	NtUserGetOpenClipboardWindow
GetPriorityClipboardFormat	0x119E	NtUserGetPriorityClipboardFormat
GetProcessWindowStation	0x119F	NtUserGetProcessWindowStation
GetScrollBarInfo	0x11A0	NtUserGetScrollBarInfo
GetSystemMenu	0x11A1	NtUserGetSystemMenu
нет	0x11A2	NtUserGetThreadDesktop
нет	0x11A3	NtUserGetThreadState

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
GetTitleBarInfo	0x11A4	NtUserGetTitleBarInfo
нет	0x11A5	NtUserGetUpdateRect
нет	0x11A6	NtUserGetUpdateRgn
GetWindowDC	0x11A7	NtUserGetWindowDC
GetWindowPlacement	0x11A8	NtUserGetWindowPlacement
нет	0x11A9	NtUserGetWOWClass
нет	0x11AA	NtUserHardErrorControl
HideCaret	0x11AB	NtUserHideCaret
HiliteMenuItem	0x11AC	NtUserHiliteMenuItem
ImpersonateDdeClientWindow	0x11AD	NtUserImpersonateDdeClientWindow
нет	0x11AE	NtUserInitialize
нет	0x11AF	NtUserInitializeClientPfnArrays
нет	0x11B0	NtUserInitTask
нет	0x11B1	NtUserInternalGetWindowText
InvalidateRect	0x11B2	NtUserInvalidateRect
InvalidateRgn	0x11B3	NtUserInvalidateRgn
IsClipboardFormatAvailable	0x11B4	NtUserIsClipboardFormatAvailable
KillTimer	0x11B5	NtUserKillTimer
нет	0x11B6	NtUserLoadKeyboardLayoutEx
LockWindowStation	0x11B7	NtUserLockWindowStation
LockWindowUpdate	0x11B8	NtUserLockWindowUpdate
LockWorkStation	0x11B9	NtUserLockWorkStation
нет	0x11BA	NtUserMapVirtualKeyEx
MenuItemFromPoint	0x11BB	NtUserMenuItemFromPoint
нет	0x11BC	NtUserMessageCall
нет	0x11BD	NtUserMinMaximize
нет	0x11BE	NtUserMNDragLeave
нет	0x11BF	NtUserMNDragOver
нет	0x11C0	NtUserMoveWindow
MoveWindow	0x11C1	NtUserMoveWindow
нет	0x11C2	NtUserNotifyIMEStatus
нет	0x11C3	NtUserNotifyProcessCreate
нет	0x11C4	NtUserNotifyWinEvent
нет	0x11C5	NtUserOpenClipboard
нет	0x11C6	NtUserOpenDesktop
OpenInputDesktop	0x11C7	NtUserOpenInputDesktop
нет	0x11C8	NtUserOpenWindowStation
PaintDesktop	0x11C9	NtUserPaintDesktop
нет	0x11CA	NtUserPeekMessage
нет	0x11CB	NtUserPostMessge
нет	0x11CC	NtUserPostThreadMessage

gdi32.dll	INT 2Eh	win32k.sys
нет	0x11CD	NtUserProcessConnect
нет	0x11CE	NtUserQueryInformationThread
нет	0x11CF	NtUserQueryInputContext
QuerySendMessage	0x11D0	NtUserQuerySendMessage
QueryUserCounters	0x11D1	NtUserQueryUserCounters
нет	0x11D2	NtUserQueryWindow
RealChildWindowFromPoint	0x11D3	NtUserRealChildWindowFromPoint
RedrawWindow	0x11D4	NtUserRedrawWindow
нет	0x11D5	NtUserRegisterClassExWOW
RegisterHotKey	0x11D6	NtUserRegisterHotKey
RegisterTasklist	0x11D7	NtUserRegisterTasklist
нет	0x11D8	NtUserRegisterWindowMessage
RemoveMenu	0x11D9	NtUserRemoveMenu
нет	0x11DA	NtUserRemoveProp
нет	0x11DB	NtUserResolveDesktop
нет	0x11DC	NtUserResolveDesktopForWOW
нет	0x11DD	NtUserSBGetParms
нет	0x11DE	NtUserScrollDC
нет	0x11DF	NtUserScrollWindowEx
нет	0x11E0	NtUserSelectPalette
SendInput	0x11E1	NtUserSendInput
нет	0x11E2	NtUserSendMessageCallback
нет	0x11E3	NtUserSendNotifyMessage
SetActiveWindow	0x11E4	NtUserSetActiveWindow
нет	0x11E5	NtUserSetAppImeLevel
SetCapture	0x11E6	NtUserSetCapture
нет	0x11E7	NtUserSetClassLong
SetClassWord	0x11E8	NtUserSetClassWord
нет	0x11E9	NtUserSetClipboardData
SetClipboardViewer	0x11EA	NtUserSetClipboardViewer
SetConsoleReserveKeys	0x11EB	NtUserSetConsoleReserveKeys
SetCursor	0x11EC	NtUserSetCursor
SetCursorContents	0x11ED	NtUserSetCursorContents
нет	0x11EE	NtUserSetCursorIconData
SetDbgTag	0x11EF	NtUserSetDbgTag
SetFocus	0x11F0	NtUserSetFocus
нет	0x11F1	NtUserSetImeHotKey
нет	0x11F2	NtUserSetImeInfoEx
нет	0x11F3	NtUserSetImeOwnerWindow
нет	0x11F4	NtUserSetInformationProcess
нет	0x11F5	NtUserSetInformationThread

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
SetInternalWindowPos	0x11F6	NtUserSetInternalWindowPos
SetKeyboardState	0x11F7	NtUserSetKeyboardState
SetLogonNotifyWindow	0x11F8	NtUserSetLogonNotifyWindow
нет	0x11F9	NtUserSetMenu
SetMenuContextHelpId	0x11FA	NtUserSetMenuContextHelpId
SetMenuDefaultItem	0x11FB	NtUserSetMenuDefaultItem
нет	0x11FC	NtUserSetMenuFlagRtoL
SetObjectInformationW	0x11FD	NtUserSetObjectInformation
SetParent	0x11FE	NtUserSetParent
SetProcessWindowStation	0x11FF	NtUserSetProcessWindowStation
нет	0x1200	NtUserSetProp
SetRipFlags	0x1201	NtUserSetRipFlags
SetScrollInfo	0x1202	NtUserSetScrollInfo
SetShellWindowEx	0x1203	NtUserSetShellWindowEx
нет	0x1204	NtUserSetSysColors
нет	0x1205	NtUserSetSystemCursor
SetSystemMenu	0x1206	NtUserSetSystemMenu
SetSystemTimer	0x1207	NtUserSetSystemTimer
SetThreadDesktop	0x1208	NtUserSetThreadDesktop
нет	0x1209	NtUserSetThreadLayoutHandles
нет	0x120A	NtUserSetThreadState
SetTimer	0x120B	NtUserSetTimer
нет	0x120C	NtUserSetWindowFNID
нет	0x120D	NtUserSetWindowLong
SetWindowPlacement	0x120E	NtUserSetWindowPlacement
SetWindowPos	0x120F	NtUserSetWindowPos
нет	0x1210	NtUserSetWindowRgn
нет	0x1211	NtUserSetWindowHookAW
нет	0x1212	NtUserSetWindowHookEx
нет	0x1213	NtUserSetWindowStationUser
SetWindowWord	0x1214	NtUserSetWindowWord
нет	0x1215	NtUserSetWinEventHook
SetShowCaret	0x1216	NtUserSetShowCaret
SetShowScrollBar	0x1217	NtUserSetShowScrollBar
SetShowWindow	0x1218	NtUserSetShowWindow
SetShowWindowAsync	0x1219	NtUserSetShowWindowAsync
нет	0x121A	NtUserSoundSentry
SwitchDesktop	0x121B	NtUserSwitchDesktop
нет	0x121C	NtUserSystemParametersInfo
нет	0x121D	NtUserTestForInteractiveUser
нет	0x121E	NtUserThunkedMenuInfo

gdi32.dll	INT 2Eh	win32k.sys
нет	0x121F	NtUserThunkedMenuItemInfo
нет	0x1220	NtUserToUnicodeEx
TrackMouseEvent	0x1221	NtUserTrackMouseEvent
TrackPopupMenuEx	0x1222	NtUserTrackPopupMenuEx
нет	0x1223	NtUserTranslateAccelerator
нет	0x1224	NtUserTranslateMessage
UnhookWindowsHookEx	0x1225	NtUserUnhookWindowsHookEx
UnhookWinEvent	0x1226	NtUserUnhookWinEvent
нет	0x1227	NtUserUnloadKeyboardLayout
UnlockWindowStation	0x1228	NtUserUnlockWindowStation
нет	0x1229	NtUserUnregisterClass
UnregisterHotKey	0x122A	NtUserUnregisterHotKey
нет	0x122B	NtUserUpdateInputContext
нет	0x122C	NtUserUpdateInstance
UpdateLayeredWindow	0x122D	NtUserUpdateLayeredWindow
SetLayeredWindowAttributes	0x122E	NtUserSetLayeredWindowAttributes
нет	0x122F	NtUserUpdatePerUserSystemParameters
UserHandleGrandAccess	0x1230	NtUserUserHandleGrandAccess
нет	0x1231	NtUserValidateHandleSecure
ValidateRect	0x1232	NtUserValidateRect
нет	0x1233	NtUserVkKeyScanEx
нет	0x1234	NtUserWaiForInputIdle
нет	0x1235	NtUserWaiForMsgAndEvent
WaitMessage	0x1236	NtUserWaitMessage
Win32PoolAllocationStats	0x1237	NtUserWin32PoolAllocationStats
WindowFromPoint	0x1238	NtUserWindowFromPoint
нет	0x1239	NtUserYieldTask
нет	0x123A	NtUserRemoteConnect
нет	0x123B	NtUserRemoteRedrawRectangle
нет	0x123C	NtUserRemoteRedrawScreen
нет	0x123D	NtUserRemoteStopScreenUpdates
нет	0x123E	NtUserCtxDisplayIOCtl
EngAssociateSurface	0x123F	NtGdiEngAssociateSurface
EngCreateBitmap	0x1240	NtGdiEngCreateBitmap
EngCreateDeviceSurface	0x1241	NtGdiEngCreateDeviceSurface
CreateDeviceBitmap	0x1242	NtGdiEngCreateDeviceBitmap
CreatePalette	0x1243	NtGdiEngCreatePalette
нет	0x1244	NtGdiEngComputeGlyphSet
EngCopyBits	0x1245	NtGdiEngCopyBits
нет	0x1246	NtGdiEngDeletePalette
EngDeleteSurface	0x1247	NtGdiEngDeleteSurface

продолжение

Таблица Б.2 (*продолжение*)

gdi32.dll	INT 2Eh	win32k.sys
EngEraseSurface	0x1248	NtGdiEngEraseSurface
EngUnlockSurface	0x1249	NtGdiEngUnlockSurface
EngLockSurface	0x124A	NtGdiEngLockSurface
EngBitBlt	0x124B	NtGdiEngBitBlt
EngStretchBlt	0x124C	NtGdiEngStretchBlt
EngPlgBlt	0x124D	NtGdiEngPlgBlt
EngMarkBandingSurface	0x124E	NtGdiEngMarkBandingSurface
EngStrokePath	0x124F	NtGdiEngStrokePath
EngFillPath	0x1250	NtGdiEngFillPath
EngStrokeAndFillPath	0x1251	NtGdiEngStrokeAndFillPath
EngPaint	0x1252	NtGdiEngPaint
EngLineTo	0x1253	NtGdiEngLineTo
EngAlphaBlend	0x1254	NtGdiEngAlphaBlend
EngGradientFill	0x1255	NtGdiEngGradientFill
EngTransparentBlt	0x1256	NtGdiEngTransparentBlt
EngTextOut	0x1257	NtGdiEngTextOut
EngStretchBltROP	0x1258	NtGdiEngStretchBltROP
XLATEOBJ_cGetPalette	0x1259	NtGdiXLATEOBJ_cGetPalette
XLATEOBJ_iXlate	0x125A	NtGdiXLATEOBJ_iXlate
XLATEOBJ_hGetColorTransform	0x125B	NtGdiXLATEOBJ_hGetColorTransform
CLIPOBJ_bEnum	0x125C	NtGdiCLIPOBJ_bEnum
CLIPOBJ_cEnumStart	0x125D	NtGdiCLIPOBJ_cEnumStart
CLIPOBJ_ppoGetPath	0x125E	NtGdiCLIPOBJ_ppoGetPath
EngDeletePath	0x125F	NtGdiEngDeletePath
EngCreateClip	0x1260	NtGdiEngCreateClip
EngDeleteClip	0x1261	NtGdiEngDeleteClip
BRUSHOBJ_ulGetBrushColor	0x1262	NtGdiBRUSHOBJ_ulGetBrushColor
BRUSHOBJ_pvAllocRbrush	0x1263	NtGdiBRUSHOBJ_pvAllocRbrush
BRUSHOBJ_pvGetRbrush	0x1264	NtGdiBRUSHOBJ_pvGetRbrush
BRUSHOBJ_hGetColorTransform	0x1265	NtGdiBRUSHOBJ_hGetColorTransform
XFORMOBJ_bApplyXform	0x1266	NtGdiXFORMOBJ_bApplyXform
XFORMOBJ_iGetXform	0x1267	NtGdiXFORMOBJ_iGetXform
FONTOBJ_vGetInfo	0x1268	NtGdiFONTOBJ_vGetInfo
FONTOBJ_pxoGetXform	0x1269	NtGdiFONTOBJ_pxoGetXform
FONTOBJ_cGetGlyphs	0x126A	NtGdiFONTOBJ_cGetGlyphs
FONTOBJ_pifi	0x126B	NtGdiFONTOBJ_pifi
FONTOBJ_pfdg	0x126C	NtGdiFONTOBJ_pfdg
FONTOBJ_QueryGlyphAttrs	0x126D	NtGdiFONTOBJ_QueryGlyphAttrs
FONTOBJ_pvTrueTypeFonFile	0x126E	NtGdiFONTOBJ_pvTrueTypeFonFile
FONTOBJ_cGetAllGlyphHandles	0x126F	NtGdiFONTOBJ_cGetAllGlyphHandles
STROBJ_bEnum	0x1270	NtGdiSTROBJ_bEnum

gdi32.dll	INT 2Eh	win32k.sys
STROBJ_bEnumPositionsOnly	0x1271	NtGdiSTROBJ_bEnumPositionsOnly
STROBJ_GetAdvanceWidth	0x1272	NtGdiSTROBJ_GetAdvanceWidth
STROBJ_vEnumStart	0x1273	NtGdiSTROBJ_vEnumStart
STROBJ_dwGetCodePage	0x1274	NtGdiSTROBJ_dwGetCodePage
PATHOBJ_vGetBounds	0x1275	NtGdiPATHOBJ_vGetBounds
PATHOBJ_bEnum	0x1276	NtGdiPATHOBJ_bEnum
PATHOBJ_vEnumStart	0x1277	NtGdiPATHOBJ_vEnumStart
PATHOBJ_vEnumStartClipLines	0x1278	NtGdiPATHOBJ_vEnumStartClipLines
PATHOBJ_bEnumClipLines	0x1279	NtGdiPATHOBJ_bEnumClipLines
нет	0x127A	NtGdiGetDhpdev
EngCheckAbort	0x127B	NtGdiEngCheckAbort
HT_Get8BPPFormatPalette	0x127C	NtGdiHT_Get8BPPFormatPalette
HT_Get8BPPMaskPalette	0x127D	NtGdi HT_Get8BPPMaskPalette
нет	0x127E	NtGdiUpdateTransform

Таблица Б.3. Библиотека времени выполнения C

Имя функции	ntdll.dll	ntoskrnl.exe
__isascii		нет
__iscsym		нет
__iscsymf		нет
__toascii		нет
_abnormal_termination	нет	
_alldiv		
_allmul		
_alloca_probe		нет
_allrem		
_allshl		
_allshr		
_atoi64		нет
_aulldiv		
_aullrem		
_aullshr		нет
_chkstk		
_CIpow		нет
_except_handler2	нет	
_except_handler3	нет	
_fltused		нет
_ftol		нет
_global_unwind2	нет	
_i64toa		нет
_i64tow		нет

продолжение

Таблица Б.3 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
_itoa		
_itow		
_local_unwind2	нет	
_ltoa		нет
_ltow		нет
_memccpy		нет
_memicmp		нет
_purecall	нет	
_snprintf		
_snwprintf		нет
_splipath		нет
_strcmpi		
_stricmp		
_strlwr		
_strnicmp		
_strnset	нет	
_strrev	нет	
_strset	нет	
_strupr		
_tolower		нет
_toupper		нет
_ui64toa		нет
_ultoa		нет
_ultow		нет
_vsnprintf		
_wcsicmp		
_wcslwr		
_wcsnicmp		
_wcsnset	нет	
_wcsrev	нет	
_wcsupr		
_wtoi		нет
_wtoi64		нет
_wtol		нет
abs		нет
atan		нет
atoi		
atol		
ceil		нет
cos		нет

Имя функции	ntdll.dll	ntoskrnl.exe
fabs		нет
floor		нет
isalnum		нет
isalpha		нет
iscntrl		нет
isdigit		
isgraph		нет
islower		
isprint		
ispunct		нет
isspace		
isupper		
iswalpha		нет
iswctype		нет
iswdigit		нет
iswlower		нет
iswspace		нет
iswxdigit		нет
isxdigit		
labs		нет
log		нет
mbstowcs		
mbtowc	нет	
memchr		нет
memcmp		
memcpy		
memmove		
memset		
pow		нет
qsort		
rand	нет	
sin		нет
sprintf		
sqrt		нет
srand	нет	
sscanf		нет
strcat		
strchr		
strcmp		
strcpy		

продолжение

Таблица Б.3 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
strcspn		нет
strlen		
strncat		
strncmp		
strncpy		
strpbrk		нет
strrchr		
strspn		
strstr		
strtol		нет
strtoul		нет
swprintf		
tan		нет
tolower		
toupper		
towlower		
towupper		
vsprintf		
wcscat		
wcschr		
wcscmp		
wcscpy		
wcscspn		
wcslen		
wcsncat		
wcsncmp		
wcsncpy		
wcspbrk		нет
wcsrchr		
wcsspn		
wcsstr		
wcstombs	нет	
wcstomb	нет	
wcstol		нет
wcstombs[1]		нет
wcstoul		нет

[1] Опять повтор значения, которое встречается выше. Точно так же и в оригинальной книге. К сожалению, мне не удалось проверить правильность этой таблицы. Я буду благодарен читателям за любую информацию на эту тему. — *Примеч. ред.*

Таблица Б.4. Библиотека времени выполнения Windows 2000

Имя функции	ntdll.dll	ntoskrnl.exe
RtlAbortRXact		нет
RtlAbsoluteToSelfRelativeSD		
RtlAcquirePebLock		нет
RtlAcquireResourceExclusive		нет
RtlAcquireShared		нет
RtlAddAccessAllowedAce		
RtlAddAccessAllowedAceEx		нет
RtlAddAccessAllowedObjectAce		нет
RtlAddAccessDeniedAce		нет
RtlAddAccessDeniedAceEx		нет
RtlAddAccessDeniedObjectAc		нет
RtlAddAce		
RtlAddActionToRXact		нет
RtlAddAtomToAtomTable		
RtlAddAttributeActionToRXact		нет
RtlAddAuditAccessAce		нет
RtlAddAuditAccessAceEx		нет
RtlAddAuditAccessObjectAce		нет
RtlAddCompoundAce		нет
RtlAddRange		
RtlAdjustPrivilege		нет
RtlAllocateAndInitializeSid		нет
RtlAllocateHandle		
RtlAllocateHeap		
RtlAnsiCharToUnicodeChar		
RtlAnsiStringToUnicodeSize		
RtlAnsiStringToUnicodeString		
RtlAppendAsciizToString		
RtlAppendStringToString		
RtlAppendUnicodeStringToString		
RtlAppendUnicodeToString		
RtlApplyRXact		нет
RtlApplyRXactNoFlash		нет
RtlAreAllAccessesGranted		
RtlAreAnyAccessesGranted		
RtlAreBitsClear		
RtlAreBitsSet		
RtlAssert		
RtlCallbackLpcClient		нет
RtlCancelTimer		нет
RtlCaptureContext	нет	

продолжение

Таблица Б.4 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
RtlCaptureStackBackTrace		
RtlCharToInteger		
RtlCheckForOrphanedCriticalSections		нет
RtlCheckRegistryKey		
RtlClearAllBits		
RtlClearBits		
RtlCompactHeap		нет
RtlCompareMemory		
RtlCompareMemoryUlong		
RtlCompareString		
RtlpCompareUnicodeString		
RtlCompressBuffer		
RtlCompressChunks	нет	
RtlConsoleMultiByteToUnicodeN		нет
RtlConvertExclusiveToShared		нет
RtlConvertLongToLargeInteger		
RtlConvertPropertyToVariant		нет
RtlConvertSharedToExclusive		нет
RtlConvertSidToUnicodeString		
RtlConvertToAutoInheritSecuretyObject		нет
RtlConvertUiListToApiList		нет
RtlConvertUlongToLargeInteger		
RtlConvertVariantToProperty		нет
RtlCopyLuid		
RtlCopyLuidAndAttributesArray		нет
RtlCopyRangeList		
RtlCopySecurityDescriptor		нет
RtlCopySid		
RtlCopySidAndAttributesArray		нет
RtlCopyString		
RtlCopyUnicodeString		
RtlCreateAcl		
RtlCreateAndSetSD		нет
RtlCreateAtomTable		
RtlCreateEnvironment		нет
RtlCreateHeap		
RtlCreateLpcServer		нет
RtlCreateProcessParameters		нет
RtlCreateQueryDebugBuffer		нет
RtlCreateRegistryKey		
RtlCreateSecurityDescriptor		

Имя функции	ntdll.dll	ntoskrnl.exe
RtlCreateTagHeap		нет
RtlCreateTimer		нет
RtlCreateTimerQueue		нет
RtlCreateUnicodeString		
RtlCreateUnicodeStringFromAsciiz		нет
RtlCreateUserProcess		нет
RtlCreateUserSecurityObject		нет
RtlCreateUserThread		нет
RtlCustomCPToUnicodeN		
RtlCutoverTimeToSystemTime		нет
RtlDebugPrintTimes		нет
RtlDecompressBuffer		
RtlDecompressChunks	нет	
RtlDecompressFragment		
RtlDefaultNpAcl		нет
RtlDelete		
RtlDeleteAce		
RtlDeleteAtomFromAtomTable		
RtlDeleteCriticalSection		нет
RtlDeleteElementGenericTable		
RtlDeleteNoSplay		
RtlDeleteOwnersRangers		
RtlDeleteRange		
RtlDeleteRegistryValue		
RtlDeleteResource		нет
RtlDeleteSecurityObject		нет
RtlDeleteTimer		нет
RtlDeleteTimerQueue		нет
RtlDeleteTimerQueueEx		нет
RtlDeNormalizeProcessParams		нет
RtlDerigisterWait		нет
RtlDerigisterWaitEx		нет
RtlDescribeChunk	нет	
RtlDestroyAtomTable		
RtlDestroyEnvironment		нет
RtlDestroyHandleTable		нет
RtlDestroyHeap		
RtlDestroyProcessParameters		нет
RtlDestroyQueryDebugBuffer		нет
RtlDetermineDosPathNameType_U		нет
RtlDnsHostNameToComputerName		нет

продолжение

Таблица Б.4 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
RtlDoesFileExists_U		нет
RtlDosPathNameToNtPathName_U		нет
RtlDosSearchPath_U		нет
RtlDowncaseUnicodeString		
RtlDumpResource		нет
RtlEmptyAtomTable		нет
RtlEnableEarlyCriticalSectionEventCreation	нет	
RtlEnlargedIntegerMultiply		
RtlEnlargedUnsignedDivide		
RtlEnlargedUnsignedMultiply		
RtlEnterCriticalSection		нет
RtlEnumerateGenericTable		
RtlEnumerateGenericTableWithoutSplaying		
RtlEnumProcessHeaps		нет
RtlEqualComputerName		нет
RtlEqualDomainName		нет
RtlEqualLuid		
RtlEqualPrefixSid		нет
RtlEqualSid		
RtlEqualString		
RtlEqualUnicodeString		
RtlEraseUnicodeString		нет
RtlExpandEnvironmentStrings_U		нет
RtlExtendedIntegerMultiply		
RtlExtendedLargeIntegerDivide		
RtlExtendedMagicDivide		
RtlExtendHeap		нет
RtlFillMemory		
RtlFillMemoryUlong		
RtlFindClearBits		
RtlFindClearBitsAndSet		
RtlFindClearRuns	нет	
RtlFindFirstRunClear	нет	
RtlFindLastBackwardRunClear		
RtlFindLeastSignificantBit		
RtlFindLongestRunClear		
RtlFindMessage		
RtlFindMostSignificantBit		
RtlFindNextForwardRunClear		
RtlFindRange		
RtlFindSetBits		

Имя функции	ntdll.dll	ntoskrnl.exe
RtlFindSetBitsAndClear		
RtlFindUnicodePrefix	нет	
RtlFistFreeAce		нет
RtlFormatCurrentUserKeyPath		
RtlFormatMessage		нет
RtlFreeAnsiString		
RtlFreeHandle		нет
RtlFreeHeap		
RtlFreeOemString		
RtlFreeRangeList		
RtlFreeSid		нет
RtlFreeUnicodeString		
RtlFreeUserThreadStack		нет
RtlGenerate8dot3Name		
RtlGetAce		
RtlGetCallersAdress		
RtlGetCompressionWorkSpaceSize		
RtlGetControlSecurityDescriptor		нет
RtlGetCurrentDirectory_U		
RtlGetDaclSecurityDescriptor		нет
RtlGetDefaultCodePage	нет	
RtlGetElementGenericTable		
RtlGetFirstRange		
RtlGetFullPathName_U		нет
RtlGetGroupSecurityDescriptor		
RtlGetLongestNtPathLength		нет
RtlGetNextRange		
RtlGetNtGlobalFlags		
RtlGetNtProductType		нет
RtlGetOwnerSecurityDescriptor		
RtlGetProcessHeaps		нет
RtlGetSaclSecurityDescriptor		
RtlGetSecurityDescriptorRMControl		нет
RtlGetUserInfoHeap		нет
RtlGetVersion		нет
RtlGUIDFromString		
RtlIdentifierAuthoritySid		нет
RtlImageDirectoryEntryToData		
RtlImageNtHeader		
RtlImageRvaToSection		нет
RtlImageRvaToVa		нет

продолжение

Таблица Б.4 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
RtlImpersonateLpcClient		нет
RtlImpersonateSelf		нет
RtlInitAnsiString		
RtlInitCodePageTable		
RtlInitializeAtomPackage		нет
RtlInitializeBitMap		
RtlInitializeContext		нет
RtlInitializeCriticalSection		нет
RtlInitializeCriticalSectionAndSpinCount		нет
RtlInitializeGenericTable		
RtlInitializeHandleTable		нет
RtlInitializeRangeList		
RtlInitializeResource		нет
RtlInitializeRXact		нет
RtlInitializeSid		
RtlInitializeUnicodePrefix	нет	
RtlInitNlsTables		нет
RtlInitString		
RtlInitUnicodeString		
RtlInsertElementGenericTable		
RtlInsertElementGenericTableFull	нет	
RtlInsertUnicodePrefix	нет	
RtlInt64ToUnicodeString		нет
RtlIntegerToChar		
RtlIntegerToUnicodeString		
RtlInvertRangeList		
RtlIsDosDeviceName_U		нет
RtlIsGenericTableEmpty		
RtlIsNameLegalDOS8Dot3		
RtlIsRangeAvailable		
RtlIsTextUnicode		нет
RtlIsValidHandle		нет
RtlIsValidIndexHandle		нет
RtlIsValidOemCharacter	нет	
RtlLargeIntegerAdd		
RtlLargeIntegerArithmeticShift		
RtlLargeIntegerDivide		
RtlLargeIntegerNegate		
RtlLargeIntegerShiftLeft		
RtlLargeIntegerShiftRight		
RtlLargeIntegerSubtract		

Имя функции	ntdll.dll	ntoskrnl.exe
RtlLargeIntegerToChar		нет
RtlLeaveCriticalSection		нет
RtlLengthRequiredSid		
RtlLengthSecurityDescriptor		
RtlLengthSid		
RtlLocalTimeToSystemTime		нет
RtlLockHeap		нет
RtlLookupAtomInAtomTable		
RtlLookupElementGenericTable		
RtlLookupElementGenericTableFull	нет	
RtlMakeSelfRelativeSD		нет
RtlMapGenericMask		
RtlMergeRangeLists		
RtlMoveMemory		
RtlMultiByteToUnicodeN		
RtlMultiByteToUnicodeSize		
RtlNewInstanceSecurityObject		нет
RtlNewSecurityGrantedAccess		нет
RtlNewSecurityObject		нет
RtlNewSecurityObjectEx		нет
RtlNextUnicodePrefix	нет	
RtlNormalizeProcessParams		нет
RtlNtStatusToDosError		
RtlNtStatusToDosErrorNoTeb	нет	
RtlNumberGenericTableElements		
RtlNumberOfClearBits		
RtlNumberOfSetBits		
RtlOemStringToCountedUnicodeString	нет	
RtlOemStringToUnicodeSize		
RtlOemStringToUnicodeString		
RtlOemToUnicodeN		
RtlOpenCurrentUser		нет
RtlPcToFileHeader		нет
RtlPinAtomInAtomTable		
RtlNtCreateKey		нет
RtlNtEnumerateSubKey		нет
RtlNtMakeTemporaryKey		нет
RtlNtOpenKey		нет
RtlNtQueryValueKeySet		нет
RtlNtSetValueKey		нет
RtlPrefixString		

продолжение

Таблица Б.4 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
RtlPrefixUnicodeString		
RtlProtectHeap		нет
RtlpUnWaitCriticalSection		нет
RtlpWaitForCriticalSection		нет
RtlQueryAtomInAtomTable		
RtlQueryEnvironmentVariable_U		нет
RtlQueryInformationAcl		нет
RtlQueryProcessBackTraceInformation		нет
RtlQueryProcessDebugInformation		нет
RtlQueryProcessHeapInformation		нет
RtlQueryProcessLockInformation		нет
RtlQueryRegistryValues		
RtlQuerySecurityObject		нет
RtlQueryTagHeap		нет
RtlQueryTimeZoneInformation		
RtlQueueWorkItem		нет
RtlRaiseException		
RtlRaiseStatus		нет
RtlRandom		
RtlReAllocateHeap		нет
RtlRealPredecessor		
RtlRealSuccessor		
RtlRegisterWait		нет
RtlReleasePebLock		нет
RtlReleaseResource		нет
RtlRemonteCall		нет
RtlRemoveUnicodePrefix	нет	
RtlReserveChunk	нет	
RtlResetRtlTranslations		нет
RtlRunDecodeUnicodeString		нет
RtlRunEncodeUnicodeString		нет
RtlSecondsSince1970ToTime		
RtlSecondsSince1980ToTime		
RtlSelfRelativeToAbsoluteSD		
RtlSelfRelativeToAbsoluteSD2		
RtlSetAllBits		
RtlSetAttributesSecurityDescriptor		нет
RtlSetBits		
RtlSetControlSecurityDescriptor		нет
RtlSetCriticalSectionSpinCount		нет
RtlSetCurrentDirectory_U		нет

Имя функции	ntdll.dll	ntoskrnl.exe
RtlSetCurrentEnvironment		нет
RtlSetDaclSecurityDescriptor		нет
RtlSetEnvironmentVariable		нет
RtlSetGroupSecurityDescriptor		
RtlSetInformationAcl		нет
RtlSetIoCompletionCallback		нет
RtlSetOwnerSecurityDescriptor		
RtlSetSaclSecurityDescriptor		
RtlSetSecurityDescriptorRMControl		нет
RtlSetSecurityObject		нет
RtlSetSecurityObjectEx		нет
RtlSetThreadPoolStartFunc		нет
RtlSetTimer		нет
RtlSetTimeZoneInformation		
RtlSetUnicodeCallouts		нет
RtlSetUserFlags		нет
RtlSetUserValueHeap		нет
RtlShutdownLpcServer		нет
RtlSizeHeap		
RtlSplay		
RtlStartRXact		нет
RtlStringFromGUID		
RtlSubAuthorityCountSid		
RtlSubAuthoritySid		
RtlSubtreePredecessor		
RtlSubtreeSuccessor		
RtlSystemTimeToLocalTime		нет
RtlTimeFieldsToTime		
RtlTimeToElapsedTimeFields		нет
RtlTimeToSecondsSince1970		
RtlTimeToSecondsSince1980		
RtlTimeToTimeFields		
RtlTryEnterCriticalSection		нет
RtlUlongByteSwap		
RtlUlonglongByteSwap		
RtlUnicodeStringToAnsiSize		
RtlUnicodeStringToAnsiString		
RtlUnicodeStringToCountedOemString		
RtlUnicodeStringToInteger		
RtlUnicodeStringToOemSize		
RtlUnicodeStringToOemString		

продолжение

Таблица Б.4 (*продолжение*)

Имя функции	ntdll.dll	ntoskrnl.exe
RtlUnicodeToCustomCPN		
RtlUnicodeToMultiByteN		
RtlUnicodeToMultiByteSize		
RtlUnicodeToOemN		
RtlUniform		нет
RtlUnlockHeap		нет
RtlUnwind		
RtlUpcaseUnicodeChar		
RtlUpcaseUnicodeString		
RtlUpcaseUnicodeStringToAnsiString		
RtlUpcaseUnicodeStringToCountedOemString		
RtlUpcaseUnicodeStringToOemString		
RtlUpcaseUnicodeToCustomCPN		
RtlUpcaseUnicodeToMultiByteN		
RtlUpcaseUnicodeToOemN		
RtlUpdateTimer		нет
RtlUpperChar		
RtlUpperString		
RtlUsageHeap		нет
RtlUshortByteSwap		
RtlValidAcl		нет
RtlValidateHeap		нет
RtlValidateProcessHeaps		нет
RtlValidRelativeSecurityDescriptor		
RtlValidSecurityDescriptor		
RtlValidSid		
RtlVerifyVersionInfo		нет
RtlVolumeDeviceToDosName	нет	
RtlWalkFrameChain		
RtlWalkHeap		нет
RtlWriteRegistryValue		
RtlxAnsiStringToUnicodeSize		
RtlxOemStringToUnicodeSize		
RtlxUnicodeStringToAnsiSize		
RtlxUnicodeStringToOemSize		
RtlZeroHeap		
RtlZeroMemory		

Константы, перечисления и структуры

Примеры кода и описания в книге часто ссылаются на определения данных из комплектов Windows 2000 Sevice Driver Kit (DDK), Win32 Platform Software Development Kit (SDK), и заголовочные файлы с компакт-диска, прилагающегося к книге. Для упрощения поиска этих определений я собрал самые важные из них в настоящее приложение. Большая часть определений взята из заголовочного файла w2k_def.h, расположенного в каталоге \src\common\include компакт-диска.

Константы

Здесь собраны определения символьных констант, встречающихся во всей книге. Они также используются и в следующих разделах этого приложения.

Коды типов объектов диспетчера

```
#define  DISP_TYPE_NOTIFICATION_EVENT          0
#define  DISP_TYPE_SYNCHRONIZATION_EVENT       1
#define  DISP_TYPE_MUTANT                      2
#define  DISP_TYPE_PROCESS                     3
#define  DISP_TYPE_QUEUE                       4
#define  DISP_TYPE_SEMAPHORE                   5
#define  DISP_TYPE_THREAD                      6
#define  DISP_TYPE_NOTIFICATION_TIMER          8
#define  DISP_TYPE_SYNCHRONIZATION_TIMER       9
```

Флаги объекта-файла

```
#define  FO_FILE_OPEN                          0x00000001
#define  FO_SYNCHRONOUS_IO                     0x00000002
#define  FO_ALERTABLE_IO                       0x00000004
#define  FO_NO_INTERMEDIATE_BUFFERING          0x00000008
```

```
#define FO_WRITE_THROUGH                0x00000010
#define FO_SEQUENTIAL_ONLY              0x00000020
#define FO_CACHE_SUPPORTED              0x00000040
#define FO_NAMED_PIPE                   0x00000080
#define FO_STREAM_FILE                  0x00000100
#define FO_MAILSLOT                     0x00000200
#define FO_GENERATE_AUDIT_ON_CLOSE      0x00000400
#define FO_DIRECT_DEVICE_OPEN           0x00000800
#define FO_FILE_MODIFIED                0x00001000
#define FO_FILE_SIZE_CHANGED            0x00002000
#define FO_CLEANUP_COMPLETE             0x00004000
#define FO_TEMPORARY_FILE               0x00008000
#define FO_DELETE_ON_CLOSE              0x00010000
#define FO_OPENED_CASE_SENSITIVE        0x00020000
#define FO_HANDLE_CREATED               0x00040000
#define FO_FILE_FAST_IO_READ            0x00080000
#define FO_RANDOM_ACCESS                0x00100000
#define FO_FILE_OPEN_CANCELLED          0x00200000
#define FO_VOLUME_OPEN                  0x00400000
```

Идентификаторы раздела каталога формата Portable Executable

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8
#define IMAGE_DIRECTORY_ENTRY_TLS             9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11
#define IMAGE_DIRECTORY_ENTRY_IAT             12
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES      16
```

Типы кодов системных структур данных ввода-вывода

```
#define IO_TYPE_ADAPTER                 1
#define IO_TYPE_CONTROLLER              2
#define IO_TYPE_DEVICE                  3
#define IO_TYPE_DRIVER                  4
#define IO_TYPE_FILE                    5
#define IO_TYPE_IRP                     6
#define IO_TYPE_MASTER_ADAPTER          7
#define IO_TYPE_OPEN_PACKET             8
```

```
#define IO_TYPE_TIMER                   9
#define IO_TYPE_VPB                     10
#define IO_TYPE_ERROR_LOG               11
#define IO_TYPE_ERROR_MESSAGE           12
#define IO_TYPE_DEVICE_OBJECT_EXTENSION 13

#define IO_TYPE_APC                     18
#define IO_TYPE_DPC                     19
#define IO_TYPE_DEVICE_QUEUE            20
#define IO_TYPE_EVENT_PAIR              21
#define IO_TYPE_INTERRUPT               22
#define IO_TYPE_PROFILE                 23
```

Функции пакета запросов ввода-вывода (IRP)

```
#define IRP_MJ_CREATE                   0
#define IRP_MJ_CREATE_NAMED_PIPE        1
#define IRP_MJ_CLOSE                    2
#define IRP_MJ_READ                     3
#define IRP_MJ_WRITE                    4
#define IRP_MJ_QUERY_INFORMATION        5
#define IRP_MJ_SET_INFORMATION          6
#define IRP_MJ_QUERY_EA                 7
#define IRP_MJ_SET_EA                   8
#define IRP_MJ_FLUSH_BUFFERS            9
#define IRP_MJ_QUERY_VOLUME_INFORMATION 10
#define IRP_MJ_SET_VOLUME_INFORMATION   11
#define IRP_MJ_DIRECTORY_CONTROL        12
#define IRP_MJ_FILE_SYSTEM_CONTROL      13
#define IRP_MJ_DEVICE_CONTROL           14
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  15
#define IRP_MJ_SHUTDOWN                 16
#define IRP_MJ_LOCK_CONTROL             17
#define IRP_MJ_CLEANUP                  18
#define IRP_MJ_CREATE_MAILSLOT          19
#define IRP_MJ_QUERY_SECURITY           20
#define IRP_MJ_SET_SECURITY             21
#define IRP_MJ_POWER                    22
#define IRP_MJ_SYSTEM_CONTROL           23
#define IRP_MJ_DEVICE_CHANGE            24
#define IRP_MJ_QUERY_QUOTA              25
#define IRP_MJ_SET_QUOTA                26
#define IRP_MJ_PNP                      27
#define IRP_MJ_MAXIMUM_FUNCTION         27

#define IRP_MJ_FUNCTIONS (IRP_MJ_MAXIMUM_FUNCTION + 1)
```

Флаги заголовка объекта

```
#define OB_FLAG_CREATE_INFO     0x01    // содержит OBJECT_CREATE_INFO
#define OB_FLAG_KERNEL_MODE     0x02    // создается ядром
#define OB_FLAG_CREATOR_INFO    0x04    // содержит OBJECT_CREATOR_INFO
#define OB_FLAG_EXCLUSIVE       0x08    // OBJ_EXCLUSIVE
```

```
#define OB_FLAG_PERMANENT       0x10    // OBJ_PERMANENT
#define OB_FLAG_SECURITY        0x20    // содержит дескриптор безопасности
#define OB_FLAG_SINGLE_PROCESS 0x40     // нет HandleDBList
```

Индексы массивов типов объектов

```
#define OB_TYPE_INDEX_TYPE                1 // [ObjT] "Type"
#define OB_TYPE_INDEX_DIRECTORY           2 // [Dire] "Directory"
#define OB_TYPE_INDEX_SYMBOLIC_LINK       3 // [Symb] "SymbolicLink"
#define OB_TYPE_INDEX_TOKEN               4 // [Toke] "Token"
#define OB_TYPE_INDEX_PROCESS             5 // [Proc] "Process"
#define OB_TYPE_INDEX_THREAD              6 // [Thre] "Thread"
#define OB_TYPE_INDEX_JOB                 7 // [Job ] "Job"
#define OB_TYPE_INDEX_EVENT               8 // [Even] "Event"
#define OB_TYPE_INDEX_EVENT_PAIR          9 // [Even] "EventPair"
#define OB_TYPE_INDEX_MUTANT             10 // [Muta] "Mutant"
#define OB_TYPE_INDEX_CALLBACK           11 // [Call] "Callback"
#define OB_TYPE_INDEX_SEMAPHORE          12 // [Sema] "Semaphore"
#define OB_TYPE_INDEX_TIMER              13 // [Time] "Timer"
#define OB_TYPE_INDEX_PROFILE            14 // [Prof] "Profile"
#define OB_TYPE_INDEX_WINDOW_STATION     15 // [Wind] "WindowStation"
#define OB_TYPE_INDEX_DESKTOP            16 // [Desk] "Desktop"
#define OB_TYPE_INDEX_SECTION            17 // [Sect] "Section"
#define OB_TYPE_INDEX_KEY                18 // [Key ] "Key"
#define OB_TYPE_INDEX_PORT               19 // [Port] "Port"
#define OB_TYPE_INDEX_WAITABLE_PORT      20 // [Wait] "WaitablePort"
#define OB_TYPE_INDEX_ADAPTER            21 // [Adap] "Adapter"
#define OB_TYPE_INDEX_CONTROLLER         22 // [Cont] "Controller"
#define OB_TYPE_INDEX_DEVICE             23 // [Devi] "Device"
#define OB_TYPE_INDEX_DRIVER             24 // [Driv] "Driver"
#define OB_TYPE_INDEX_IO_COMPLETION      25 // [IoCo] "IoCompletion"
#define OB_TYPE_INDEX_FILE               26 // [File] "File"
#define OB_TYPE_INDEX_WMI_GUID           27 // [WmiG] "WmiGuid"
```

Теги типов объектов

```
#define OB_TYPE_TAG_TYPE                'TjbO' // [ObjT] "Type"
#define OB_TYPE_TAG_DIRECTORY           'eriD' // [Dire] "Directory"
#define OB_TYPE_TAG_SYMBOLIC_LINK       'bmyS' // [Symb] "SymbolicLink"
#define OB_TYPE_TAG_TOKEN               'ekoT' // [Toke] "Token"
#define OB_TYPE_TAG_PROCESS             'corP' // [Proc] "Process"
#define OB_TYPE_TAG_THREAD              'erhT' // [Thre] "Thread"
#define OB_TYPE_TAG_JOB                 ' boJ' // [Job ] "Job"
#define OB_TYPE_TAG_EVENT               'nevE' // [Even] "Event"
#define OB_TYPE_TAG_EVENT_PAIR          'nevE' // [Even] "EventPair"
#define OB_TYPE_TAG_MUTANT              'atuM' // [Muta] "Mutant"
#define OB_TYPE_TAG_CALLBACK            'llaC' // [Call] "Callback"
#define OB_TYPE_TAG_SEMAPHORE           'ameS' // [Sema] "Semaphore"
#define OB_TYPE_TAG_TIMER               'emiT' // [Time] "Timer"
#define OB_TYPE_TAG_PROFILE             'forP' // [Prof] "Profile"
#define OB_TYPE_TAG_WINDOW_STATION      'dniW' // [Wind] "WindowStation"
#define OB_TYPE_TAG_DESKTOP             'kseD' // [Desk] "Desktop"
#define OB_TYPE_TAG_SECTION             'tceS' // [Sect] "Section"
#define OB_TYPE_TAG_KEY                 ' yeK' // [Key ] "Key"
#define OB_TYPE_TAG_PORT                'troP' // [Port] "Port"
```

```
#define OB_TYPE_TAG_WAITABLE_PORT       'tiaW  // [Wait] "WaitablePort"
#define OB_TYPE_TAG_ADAPTER             'padA' // [Adap] "Adapter"
#define OB_TYPE_TAG_CONTROLLER          'tnoC' // [Cont] "Controller"
#define OB_TYPE_TAG_DEVICE              'iveD' // [Devi] "Device"
#define OB_TYPE_TAG_DRIVER              'virD' // [Driv] "Driver"
#define OB_TYPE_TAG_IO_COMPLETION       'oCoI' // [IoCo] "IoCompletion"
#define OB_TYPE_TAG_FILE                'eliF' // [File] "File"
#define OB_TYPE_TAG_WMI_GUID            'GimW' // [WmiG] "WmiGuid"
```

Флаги атрибутов объекта

```
#define OBJ_INHERIT                     0x00000002
#define OBJ_PERMANENT                   0x00000010
#define OBJ_EXCLUSIVE                   0x00000020
#define OBJ_CASE_INSENSITIVE            0x00000040
#define OBJ_OPENIF                      0x00000080
#define OBJ_OPENLINK                    0x00000100
#define OBJ_KERNEL_HANDLE               0x00000200
#define OBJ_VALID_ATTRIBUTES            0x000003F2
```

Перечисления

Некоторые константы Windows 2000 представлены в виде перечислений. Здесь приведены определения наиболее часто используемых перечислений в алфавитном порядке. Значения членов перечислений показаны в комментариях перед каждой строкой определения.

IO_ALLOCATION_ACTION

```
typedef enum _IO_ALLOCATION_ACTION
        {
/*001*/ KeepObject = 1,
/*002*/ DeallocateObject,
/*003*/ DeallocateObjectKeepRegisters
        }
        IO_ALLOCATION_ACTION;
```

LOOKASIDE_LIST_ID

```
typedef enum _LOOKASIDE_LIST_ID
        {
/*000*/ SmallIrpLookasideList,
/*001*/ LargeIrpLookasideList,
/*002*/ MdlLookasideList,
/*003*/ CreateInfoLookasideList,
/*004*/ NameBufferLookasideList,
/*005*/ TwilightLookasideList,
/*006*/ CompletionLookasideList
        }
        LOOKASIDE_LIST_ID;
```

MODE (см. также KPROCESSOR_MODE)

```
typedef enum _MODE
        {
/*000*/ KernelMode,
/*001*/ UserMode,
/*002*/ MaximumMode
        }
        MODE;
```

NT_PRODUCT_TYPE

```
typedef enum _NT_PRODUCT_TYPE
        {
/*000*/ NtProductInvalid,
/*001*/ NtProductWinNt,
/*002*/ NtProductLanManNt,
/*003*/ NtProductServer
        }
        NT_PRODUCT_TYPE;
```

POOL_TYPE

```
typedef enum _POOL_TYPE
        {
/*000*/ NonPagedPool,
/*001*/ PagedPool,
/*002*/ NonPagedPoolMustSucceed,
/*003*/ DontUseThisType,
/*004*/ NonPagedPoolCacheAligned,
/*005*/ PagedPoolCacheAligned,
/*006*/ NonPagedPoolCacheAlignedMustS,
/*007*/ MaxPoolType
        }
        POOL_TYPE;
```

Структуры и псевдонимы

В этом разделе в алфавитном порядке приведены определения структур и псевдонимов, используемых в драйверах режима ядра и системными программами низкого уровня. Не все из них документированы. В комментариях перед именами членов структуры показаны их смещения от базового адреса структуры, что помогает читать их значения в шестнадцатеричном снимке памяти.

ANSI_STRING

```
typedef STRING ANSI_STRING;
```

CALLBACK_OBJECT

```
typedef struct _CALLBACK_OBJECT
        {
/*000*/ DWORD        Tag; // 0x6C6C6143 ("Call")
```

```
/*004*/ KSPIN_LOCK   Lock;
/*008*/ LIST_ENTRY   CallbackList;
/*010*/ BOOLEAN      AllowMultipleCallbacks;
/*014*/ }
        CALLBACK_OBJECT;
```

CLIENT_ID

```
typedef struct _CLIENT_ID
        {
/*000*/ HANDLE UniqueProcess;
/*004*/ HANDLE UniqueThread;
/*008*/ }
        CLIENT_ID;
```

CONTEXT

```
// базовый адрес 0xFFDFF13C

#define MAXIMUM_SUPPORTED_EXTENSION 512

typedef struct _CONTEXT
        {
/*000*/ DWORD        ContextFlags;
/*004*/ DWORD        Dr0;
/*008*/ DWORD        Dr1;
/*00C*/ DWORD        Dr2;
/*010*/ DWORD        Dr3;
/*014*/ DWORD        Dr6;
/*018*/ DWORD        Dr7;
/*01C*/ FLOATING_SAVE_AREA FloatSave;
/*08C*/ DWORD        SegGs;
/*090*/ DWORD        SegFs;
/*094*/ DWORD        SegEs;
/*098*/ DWORD        SegDs;
/*09C*/ DWORD        Edi;
/*0A0*/ DWORD        Esi;
/*0A4*/ DWORD        Ebx;
/*0A8*/ DWORD        Edx;
/*0AC*/ DWORD        Ecx;
/*0B0*/ DWORD        Eax;
/*0B4*/ DWORD        Ebp;
/*0B8*/ DWORD        Eip;
/*0BC*/ DWORD        SegCs;
/*0C0*/ DWORD        EFlags;
/*0C4*/ DWORD        Esp;
/*0C8*/ DWORD        SegSs;
/*0CC*/ BYTE         ExtendedRegisters  [MAXIMUM_SUPPORTED_EXTENSION];
/*2CC*/ }
        CONTEXT;
```

CONTROLLER_OBJECT

```
typedef struct _CONTROLLER_OBJECT
        {
/*000*/ SHORT                Type; // IO_TYPE_CONTROLLER 0x02
/*002*/ SHORT                Size; // количество BYTE
```

```
/*004*/ PVOID                ControllerExtension;
/*008*/ KDEVICE_QUEUE        DeviceWaitQueue;
/*01C*/ DWORD                Spare1;
/*020*/ LARGE_INTEGER        Spare2;
/*028*/ }
        CONTROLLER_OBJECT;
```

CRITICAL_SECTION

```
typedef struct _RTL_CRITICAL_SECTION
        {
/*000*/ PRTL_CRITICAL_SECTION_DEBUG     DebugInfo;
/*004*/ LONG                            LockCount;
/*008*/ LONG                            RecursionCount;
/*00C*/ HANDLE                          OwningThread;
/*010*/ HANDLE                          LockSemaphore;
/*014*/ DWORD_PTR                       SpinCount;
/*018*/ }
        CRITICAL_SECTION;
```

DEVICE_OBJECT

```
typedef struct _DEVICE_OBJECT
        {
/*000*/ SHORT                       Type; // IO_TYPE_DEVICE 0x03
/*002*/ WORD                        Size; // количество BYTE
/*004*/ LONG                        ReferenceCount;
/*008*/ struct _DRIVER_DBJECT      *DriverObject;
/*00C*/ struct _DEVICE_OBJECT      *NextDevice;
/*010*/ struct _DEVICE_OBJECT      *AttachedDevice;
/*014*/ struct _IRP                *CurrentIrp;
/*018*/ struct _PIO_TIMER          *Timer;
/*01C*/ DWORD                       Flags;            // DO_*
/*020*/ DWORD                       Characteristics;  // FILE_*
/*024*/ PVPB                        Vpb;
/*028*/ PVDID                       DeviceExtension;
/*02C*/ DEVICE_TYPE                 DeviceType;
/*030*/ CHAR                        StackSize;
/*034*/ union
            {
/*034*/       LIST_ENTRY            ListEntry;
/*034*/       WAIT_CONTEXT_BLOCK    Wcb;
/*05C*/       }                     Queue;
/*05C*/ DWORD                       AlignmentRequirement;
/*060*/ KDEVICE_QUEUE               DeviceQueue;
/*074*/ KDPC                        Dpc;
/*094*/ DWORD                       ActiveThreadCount;
/*098*/ PSECURITY_DESCRIPTOR        SecurityDescriptor;
/*09C*/ KEVENT                      DeviceLock;
/*0AC*/ WORD                        SectorSize;
/*0AE*/ WORD                        Spare1;
/*0B0*/ struct _DEVOBJ_EXTENSION   *DeviceObjectExtension;
/*0B4*/ PVOID                       Reserved;
/*0B8*/ }
        OEVICE_OBJECT;
```

DEVOBJ_EXTENSION

```
typedef struct _DEVOBJ_EXTENSION
        {
/*000*/ SHORT             Type; // IO_TYPE_DEVICE_OBJECT_EXTENSION 0x0D
/*002*/ WORD              Size; // количество BYTE
/*004*/ PDEVICE_OBJECT    DeviceObject;
/*008*/ }
        DEVOBJ_EXTENSION;
```

DISPATCHER_HEADER

```
typedef struct _DISPATCHER_HEADER
        {
/*000*/ BYTE         Type;          // DISP_TYPE_*
/*001*/ BYTE         Absolute;
/*002*/ BYTE         Size;          // количество DWORD
/*003*/ BYTE         Inserted;
/*004*/ LONG         SignalState;
/*00B*/ LIST_ENTRY   WaitListHead;
/*010*/ }
        DISPATCHER_HEADER;
```

DRIVER_EXTENSION

```
typedef struct _DRIVER_EXTENSION
        {
/*000*/ struct _DRIVER_OBJECT`  *DriverObject;
/*004*/ PDRIVER_ADD_DEVICE       AddDevice;
/*008*/ DWDRD                    Count;
/*00C*/ UNICODE_STRING           ServiceKeyName;
/*014*/ }
        DRIVER_EXTENSION;
```

DRIVER_OBJECT

```
typedef struct _DRIVER_OBJECT
        {
/*000*/ SHORT                  Type; // IO_TYPE_DRIVER 0x04
/*002*/ SHORT                  Size; // количество BYTE
/*004*/ PDEVICE_OBJECT         DeviceObject;
/*008*/ DWORD                  Flags;
/*00C*/ PVOID                  DriverStart;
/*010*/ DWORD                  DriverSize;
/*014*/ PVOID                  DriverSection;
/*018*/ PDRIVER_EXTENSION      DriverExtension;
/*01C*/ UNICODE_STRING         DriverName.
/*024*/ PUNICODE_STRING        HardwareDatabase;
/*028*/ PFAST_IO_DISPATCH      FastIoDispatch;
/*02C*/ PDRIVER_INITIALIZE     OriverInit;
/*030*/ PDRIVER_STARTIO        DriverStartIo;
/*034*/ PDRIVER_UNLOAD         DriverUnload;
/*038*/ PDRIVER_DISPATCH       MajorFunction  [IRP_MJ_FUNCTIONS];
/*0A8*/ }
        DRIVER_OBJECT;
```

EPROCESS

```
typedef struct _EPROCESS
        {
/*000*/ KPROCESS                Pcb.
/*06C*/ NTSTATUS                ExitStatus;
/*070*/ KEVENT                  LockEvent;
/*080*/ DWORD                   LockCount;
/*084*/ DWORD                   d084.
/*088*/ LARGE_INTEGER           CreateTime;
/*090*/ LARGE_INTEGER           ExitTime;
/*098*/ PVOID                   LockOwner.
/*09C*/ DWORD                   UniqueProcessId;
/*0A0*/ LIST_ENTRY              ActiveProcessLinks;
/*0A8*/ DWORD                   QuotaPeakPoolUsage [2]. // NP. P
/*0B0*/ DWORD                   QuotaPoolUsage     [2]; // NP. P
/*0B8*/ DWORD                   PagefileUsage;
/*0BC*/ DWORD                   CommitCharge.
/*0C0*/ DWORD                   PeakPagefileUsage;
/*0C4*/ DWORD                   PeakVirtualSize.
/*0C8*/ LARGE_INTEGER           VirtualSize;
/*0D0*/ MMSUPPORT               Vm;
/*100*/ DWORD                   d100.
/*104*/ DWORD                   d104.
/*108*/ DWORD                   d108;
/*10C*/ DWORD                   d10C;
/*110*/ DWORD                   d110;
/*114*/ DWORD                   d114;
/*118*/ DWORD                   d118;
/*11C*/ DWORD                   d11C;
/*120*/ PVDID                   DebugPort.
/*124*/ PVOID                   ExceptionPort;
/*128*/ PHANDLE_TABLE           ObjectTable;
/*12C*/ PVOID                   Token;
/*130*/ FAST_MUTEX              WorkingSetLock;
/*150*/ DWORD                   WorkingSetPage.
/*154*/ BOOLEAN                 ProcessOutswapEnabled;
/*155*/ BOOLEAN                 ProcessOutswapped.
/*156*/ BOOLEAN                 AddressSpaceInitialized;
/*157*/ BOOLEAN                 AddressSpaceDeleted;
/*158*/ FAST_MUTEX              AddressCreationLock;
/*178*/ KSPIN_LOCK              HyperSpaceLock.
/*17C*/ DWORD                   ForkInProgress;
/*180*/ WORD                    VmOperation.
/*182*/ BOOLEAN                 ForkWasSuccessful;
/*183*/ BYTE                    MmAgressiveWsTrimMask;
/*184*/ DWORD                   VmOperationEvent;
/*188*/ HARDWARE_PTE            PageDirectoryPte;
/*18C*/ DWORD                   LastFaultCount;
/*190*/ DWORD                   ModifiedPageCount;
/*194*/ PVDID                   VadRoot.
/*198*/ PVOID                   VadHint.
/*19C*/ PVOID                   CloneRoot;
/*1A0*/ DWORD                   NumberOfPrivatePages;
/*1A4*/ DWORD                   NumberOfLockedPages;
/*1A8*/ WORD                    NextPageColor.
/*1AA*/ BOOLEAN                 ExitProcessCalled;
```

```
/*1AB*/ BOOLEAN                 CreateProcessReported.
/*1AC*/ HANDLE                  SectionHandle.
/*1B0*/ struct _PEB             *Peb:
/*1B4*/ PVOID                   SectionBaseAddress:
/*1B8*/ PQUOTA_BLOCK            QuotaBlock:
/*1BC*/ NTSTATUS                LastThreadExitStatus:
/*1C0*/ DWORD                   WorkingSetWatch:
/*1C4*/ HANDLE                  Win32WindowStation:
/*1C8*/ DWORD                   InheritedFromUniqueProcessId:
/*1CC*/ ACCESS_MASK             GrantedAccess:
/*1D0*/ DWORD                   DefaultHardErrorProcessing: // HEM_*
/*1D4*/ DWORD                   LdtInformation:
/*1D8*/ PVOID                   VadFreeHint.
/*1DC*/ DWORD                   VdmObjects.
/*1E0*/ PVOID                   DeviceMap: // 0x24 bytes
/*1E4*/ DWORD                   SessionId:
/*1E8*/ DWORD                   d1E8:
/*1EC*/ DWORD                   d1EC:
/*1F0*/ DWORD                   d1F0:
/*1F4*/ DWORD                   d1F4:
/*1F8*/ DWORD                   d1F8:
/*1FC*/ BYTE                    ImageFileName [16]:
/*20C*/ DWORD                   VmTrimFaultValue:
/*210*/ BYTE                    SetTimerResolution:
/*211*/ BYTE                    PriorityClass:
/*212*/ union
          {
          struct
               {
/*212*/          BYTE          SubSystemMinorVersion.
/*213*/          BYTE          SubSystemMajorVersion
               }:
               struct
                    {
/*212*/               WORD      SubSystemVersion.
                    }:
          }:
/*214*/ struct _WIN32_PROCESS   *Win32Process.
/*218*/ DWORD                   d218:
/*21C*/ DWORD                   d21C:
/*220*/ DWORD                   d220:
/*224*/ DWORD                   d224:
/*228*/ DWORD                   d228:
/*22C*/ DWORD                   d22C:
/*230*/ PVOID                   Wow64:
/*234*/ DWORD                   d234:
/*238*/ IO_COUNTERS             IoCounters.
/*268*/ DWORD                   d268:
/*26C*/ DWORD                   d26C:
/*270*/ DWORD                   d270:
/*274*/ DWORD                   d274:
/*278*/ DWORD                   d278:
/*27C*/ DWORD                   d27C:
/*280*/ DWORD                   d280:
/*2B4*/ DWORD                   d284:
/*288*/ }
        EPROCESS.
```

ERESOURCE

```
typedef struct _ERESOURCE
        {
/*000*/ LIST_ENTRY          SystemResourcesList.
/*008*/ POWNER_ENTRY        OwnerTable:
/*00C*/ SHORT               ActiveCount:
/*00E*/ WORD                Flag.
/*010*/ PKSEMAPHORE         SharedWaiters:
/*014*/ PKEVENT             ExclusiveWaiters:
/*018*/ OWNER_ENTRY         OwnerThreads [2]:
/*028*/ DWORD               ContentionCount:
/*02C*/ WORD                NumberOfSharedWaiters:
/*02E*/ WDRD                NumberOfExclusiveWaiters:
/*030*/ union
            {
/*030*/     PVOID           Address
/*030*/     DWORD_PTR       CreatorBackTraceIndex:
/*034*/     }:
/*034*/ KSPIN_LOCK          SpinLock:
/*038*/ }
        ERESOURCE:
```

ERESOURCE_OLD

```
typedef struct _ERESOURCE_OLD
        {
/*000*/ LIST_ENTRY              SystemResourcesList:
/*008*/ PERESOURCE_THREAD       OwnerThreads:
/*00C*/ PBYTE                   OwnerCounts.
/*010*/ WDRD                    TableSize.
/*012*/ WDRD                    ActiveCount:
/*014*/ WORD                    Flag.
/*016*/ WORD                    TableRover:
/*018*/ BYTE                    InitialOwnerCounts [4].
/*01C*/ ERESOURCE_THREAD        InitialOwnerThreads [4].
/*02C*/ DWORD                   Spare1.
/*030*/ DWORD                   ContentionCount:
/*034*/ WORD                    NumberOfExclusiveWaiters.
/*036*/ WORD                    NumberOfSharedWaiters.
/*038*/ KSEMAPHORE              SharedWaiters.
/*04C*/ KEVENT                  ExclusiveWaiters.
/*05C*/ KSPIN_LOCK              SpinLock.
/*060*/ DWORD                   CreatorBackTraceIndex:
/*064*/ WORD                    Depth
/*066*/ WORD                    Reserved:
/*068*/ PVDID                   OwnerBackTrace [4]:
/*078*/ }
        ERESOURCE_OLD.
```

ERESOURCE_THREAD

```
typedef DWDRD_PTR ERESDURCE_THREAD
```

ETHREAD

```
typedef struct _ETHREAD
        {
/*000*/ KTHREAD                 Tcb.
/*1B0*/ LARGE_INTEGER           CreateTime;
/*1B8*/ union
            {
/*1B8*/     LARGE_INTEGER       ExitTime;
/*1BB*/     LIST_ENTRY          LpcReplyChain;
            };
/*1C0*/ union
            {
/*1C0*/     NTSTATUS            ExitStatus;
/*1C0*/     DWORD               OfsChain;
            };
/*1C4*/ LIST_ENTRY              PostBlockList;
/*1CC*/ LIST_ENTRY              TerminationPortList;
/*1D4*/ PVOID                   ActiveTimerListLock;
/*1D8*/ LIST_ENTRY              ActiveTimerListHead;
/*1E0*/ CLIENT_ID               Cid.
/*1E8*/ KSEMAPHORE              LpcReplySemaphore.
/*1FC*/ DWORD                   LpcReplyMessage
/*200*/ DWORD                   LpcReplyMessageId.
/*204*/ DWORD                   PerformanceCountLow;
/*208*/ QWORD                   ImpersonationInfo.
/*20C*/ LIST_ENTRY              IrpList.
/*214*/ PVOID                   TopLevelIrp.
/*21B*/ PVDID                   DeviceToVerify.
/*21C*/ DWORD                   ReadClusterSize;
/*220*/ BOOLEAN                 ForwardClusterOnly.
/*221*/ BOOLEAN                 DisablePageFaultClustering;
/*222*/ BOOLEAN                 DeadThread
/*223*/ BOOLEAN                 Reserved
/*224*/ BOOL                    HasTerminated.
/*228*/ ACCESS_MASK             GrantedAccess.
/*22C*/ PEPROCESS               ThreadsProcess;
/*230*/ PVOID                   StartAddress;
/*234*/ union
            {
/*234*/     PVOID               Win32StartAddress.
/*234*/     DWORD               LpcReceivedMessageId;
            }.
/*238*/ BOOLEAN                 LpcExitThreadCalled;
/*239*/ BOOLEAN                 HardErrorsAreDisabled.
/*23A*/ BOOLEAN                 LpcReceivedMsgIdValid.
/*23B*/ BOOLEAN                 ActiveImpersonationInfo;
/*23C*/ DWORD                   PerformanceCountHigh
/*240*/ DWDRD                   d240.
/*244*/ DWORD                   d244.
/*24B*/ }
        ETHREAD.
```

ETIMER

```
typedef struct _ETIMER
        {
/*000*/ KTIMER       Tcb;
/*028*/ KAPC         Apc;
/*058*/ KDPC         Dpc;
/*078*/ LIST_ENTRY   ActiveTimerList;
/*080*/ KSPIN_LOCK   Lock;
/*084*/ LONG         Period;
/*088*/ BOOLEAN      Active;
/*089*/ BOOLEAN      Resume;
/*08C*/ LIST_ENTRY   WakeTimerList;
/*098*/ }
        ETIMER;
```

FILE_OBJECT

```
typedef struct _FILE_OBJECT
        {
/*000*/ SHORT                      Type; // IO_TYPE_FILE 0x05
/*002*/ SHORT                      Size; // количество BYTE
/*004*/ PDEVICE_OBJECT             DeviceObject;
/*008*/ PVPB                       Vpb;
/*00C*/ PVOID                      FsContext;
/*010*/ PVOID                      FsContext2;
/*014*/ PSECTION_OBJECT_POINTERS   SectionObjectPointer;
/*018*/ PVOID                      PrivateCacheMap;
/*01C*/ NTSTATUS                   FinalStatus;
/*020*/ struct _FILE_OBJECT        *RelatedFileObject;
/*024*/ BOOLEAN                    LockOperation;
/*025*/ BOOLEAN                    DeletePending;
/*026*/ BOOLEAN                    ReadAccess;
/*027*/ BOOLEAN                    WriteAccess;
/*028*/ BOOLEAN                    DeleteAccess;
/*029*/ BOOLEAN                    SharedRead;
/*02A*/ BOOLEAN                    SharedWrite;
/*02B*/ BOOLEAN                    SharedDelete;
/*02C*/ DWORD                      Flags; // FO_*
/*030*/ UNICODE_STRING             FileName;
/*038*/ LARGE_INTEGER              CurrentByteOffset;
/*040*/ DWORD                      Waiters;
/*044*/ DWORD                      Busy;
/*048*/ PVOID                      LastLock;
/*04C*/ KEVENT                     Lock;
/*05C*/ KEVENT                     Event;
/*06C*/ PIO_COMPLETION_CONTEXT     CompletionContext;
/*070*/ }
        FILE_OBJECT;
```

FLOATING_SAVE_AREA

```
// базовый адрес 0xFFDFF158

typedef struct _FLOATING_SAVE_AREA
        {
```

```
/*000*/ DWORD ControlWord;
/*004*/ DWORD StatusWord;
/*008*/ DWORD TagWord.
/*00C*/ DWORD ErrorOffset;
/*010*/ DWORD ErrorSelector;
/*014*/ DWORD DataOffset;
/*018*/ DWORD DataSelector;
/*01C*/ BYTE  RegisterArea [SIZE_OF_803B7_REGISTERS];
/*06C*/ DWORD Cr0NpxState;
/*070*/ }
        FLOATING_SAVE_AREA;
```

HANDLE_ENTRY

```
#define HANDLE_ATTRIBUTE_INHERIT   0x00000002
#define HANDLE_ATTRIBUTE_MASK      0x00000007
#define HANDLE_OBJECT_MASK         0xFFFFFFF8

typedef struct _HANDLE_ENTRY // см. OBJECT_HANDLE_INFORMATION
        {
/*000*/ union
            {
/*000*/      DWORD                HandleAttributes:   // HANDLE_ATTRIBUTE_MASK
/*000*/      POBJECT_HEADER       ObjectHeader;       // HANDLE_OBJECT_MASK
/*004*/      };
/*004*/ union
            {
/*004*/      ACCESS_MASK  GrantedAccess:      // если запись используется
/*004*/      DWORD        NextEntry:          // если запись свободна
/*008*/      };
/*008*/ }
        HANDLE_ENTRY;
```

HANDLE_LAYER1, HANDLE_LAYER2, HANDLE_LAYER3

```
#define HANDLE_LAYER1_SIZE 0x00000100

typedef struct _HANDLE_LAYER1
        {
/*000*/ PHANDLE_LAYER2 Layer2 [HANDLE_LAYER_SIZE]; // биты с 18 по 25
/*400*/ }
        HANDLE_LAYER1.

typedef struct _HANDLE_LAYER2
        {
/*000*/ PHANDLE_LAYER3 Layer3 [HANDLE_LAYER_SIZE]; // биты с 10 по 17
/*400*/ }
        HANDLE_LAYER2:

typedef struct _HANDLE_LAYER3
        {
/*000*/ HANDLE_ENTRY Entries [HANDLE_LAYER_SIZE]; // биты с 2 по 9
/*800*/ }
        HANDLE_LAYER3:
```

HANDLE_TABLE

```
typedef struct _HANDLE_TABLE
        {
/*000*/ DWORD                  Reserved;
/*004*/ DWORD                  HandleCount;
/*008*/ PHANDLE_LAYER1         Layer1.
/*00C*/ struct _EPROCESS       *Process; // передается в PsChargePoolQuota ()
/*010*/ HANDLE                 UniqueProcessId;
/*014*/ DWORD                  NextEntry;
/*018*/ DWORD                  TotalEntries;
/*01C*/ ERESOURCE              HandleTableLock;
/*054*/ LIST_ENTRY             HandleTableList;
/*05C*/ KEVENT                 Event;
/*06C*/ }
        HANDLE_TABLE;
```

HARDWARE_PTE

```
typedef struct _HARDWARE_PTE
          {
/*000*/    unsigned Valid              :  1;
           unsigned Write              :  1;
           unsigned Owner              :  1;
           unsigned WriteThrough       :  1;
           unsigned CacheDisable       :  1;
           unsigned Accessed           :  1;
           unsigned Dirty              :  1;
           unsigned LargePage          :  1;
/*001*/    unsigned Global             :  1;
           unsigned CopyOnWrite        :  1;
           unsigned Prototype          :  1;
           unsigned reserved           :  1;
           unsigned PageFrameNumber    :  20;
/*004*/    }
           HARDWARE_PTE;
```

IMAGE_DATA_DIRECTORY

```
typedef struct _IMAGE_DATA_DIRECTORY
        {
/*000*/ DWORD VirtualAddress;
/*004*/ DWORD Size;
/*008*/ }
        IMAGE_DATA_DIRECTORY;
```

IMAGE_EXPORT_DIRECTORY

```
typedef struct _IMAGE_EXPORT_DIRECTORY
        {
/*000*/ DWORD   Characteristics;
/*004*/ DWORD   TimeDateStamp;
/*008*/ WORD    MajorVersion;
/*00A*/ WORD    MinorVersion;
/*00C*/ DWORD   Name.
```

```
/*010*/ DWORD   Base
/*014*/ DWORD   NumberOfFunctions;
/*018*/ DWORD   NumberOfNames;
/*01C*/ DWORD   AddressOfFunctions;
/*020*/ DWORD   AddressOfNames.
/*024*/ DWORD   AddressOfNameOrdinals;
/*028*/ }
        IMAGE_EXPORT_DIRECTORY.
```

IMAGE_FILE_HEADER

```
typedef struct _IMAGE_FILE_HEADER
        {
/*000*/ WORD    Machine;
/*002*/ WORD    NumberOfSections;
/*004*/ DWORD   TimeDateStamp.
/*008*/ DWORD   PointerToSymbolTable;
/*00C*/ DWORD   NumberOfSymbols;
/*010*/ WORD    SizeOfOptionalHeader;
/*012*/ WORD    Characteristics;
/*014*/ }
        IMAGE_FILE_HEADER;
```

IMAGE_NT_HEADERS

```
typedef struct _IMAGE_NT_HEADERS
        {
/*000*/ DWORD                   Signature;
/*004*/ IMAGE_FILE_HEADER       FileHeader;
/*018*/ IMAGE_OPTIONAL_HEADER   OptionalHeader;
/*0FB*/ }
        IMAGE_NT_HEADERS;
```

IMAGE_OPTIONAL_HEADER

```
#define  IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16

typedef struct _IMAGE_OPTIONAL_HEADER
        {
/*000*/ WORD                    Magic.
/*002*/ BYTE                    MajorLinkerVersion;
/*003*/ BYTE                    MinorLinkerVersion;
/*004*/ DWORD                   SizeOfCode;
/*008*/ DWORD                   SizeOfInitializedData;
/*00C*/ DWORD                   SizeOfUninitializedData;
/*010*/ DWORD                   AddressOfEntryPoint;
/*014*/ DWORD                   BaseOfCode;
/*018*/ DWORD                   BaseOfData;
/*01C*/ DWORD                   ImageBase;
/*020*/ DWORD                   SectionAlignment.
/*024*/ DWORD                   FileAlignment;
/*028*/ WORD                    MajorOperatingSystemVersion.
/*02A*/ WORD                    MinorOperatingSystemVersion;
/*02C*/ WORD                    MajorImageVersion;
/*02E*/ WORD                    MinorImageVersion;
```

```
/*030*/ WORD                    MajorSubsystemVersion.
/*032*/ WORD                    MinorSubsystemVersion.
/*034*/ DWORD                   Win32VersionValue.
/*038*/ DWORD                   SizeOfImage.
/*03C*/ DWORD                   SizeOfHeaders;
/*040*/ DWORD                   CheckSum
/*044*/ WORD                    Subsystem
/*046*/ WORD                    DllCharacteristics.
/*048*/ DWORD                   SizeOfStackReserve;
/*04C*/ DWORD                   SizeOfStackCommit;
/*050*/ DWORD                   SizeOfHeapReserve.
/*054*/ DWORD                   SizeOfHeapCommit;
/*058*/ DWORD                   LoaderFlags
/*05C*/ DWORD                   NumberOfRvaAndSizes.
/*060*/ IMAGE_DATA_DIRECTORY    DataDirectory
                                [IMAGE_NUMBEROF_DIRECTORY_ENTRIES].
/*0E0*/ }
        IMAGE_OPTIONAL_HEADER.
```

IO_COMPLETION

```
typedef struct _IO_COMPLETION
        {
/*000*/ KQUEUE Queue.
/*028*/ }
        IO_COMPLETION;
```

IO_COMPLETION_CONTEXT

```
typedef struct _IO_COMPLETION_CONTEXT
        {
/*000*/ PVOID Port;
/*004*/ PVOID Key;
/*008*/ }
        IO_COMPLETION_CONTEXT.
```

IO_ERROR_LOG_ENTRY

```
typedef struct _IO_ERROR_LOG_ENTRY
        {
/*000*/ SHORT                   Type. // IO_TYPE_ERROR_LOG 0x0B
/*002*/ SHORT                   Size. // количество BYTE
/*004*/ LIST_ENTRY              ErrorLogList.
/*00C*/ PDEVICE_OBJECT          DeviceObject.
/*010*/ PDRIVER_OBJECT          DriverObject.
/*014*/ DWORD                   Reserved.
/*018*/ LARGE_INTEGER           TimeStamp
/*020*/ IO_ERROR_LOG_PACKET     EntryData.
/*050*/ }
        IO_ERROR_LOG_ENTRY.
```

IO_ERROR_LOG_MESSAGE

```
typedef struct _IO_ERROR_LOG_MESSAGE
        {
/*000*/ WORD                    Type. // IO_TYPE_ERROR_MESSAGE 0x0C
```

```
/*002*/ WORD                    Size  // количество BYTE
/*004*/ WORD                    DriverNameLength.
/*008*/ LARGE_INTEGER           TimeStamp.
/*010*/ DWORD                   DriverNameOffset.
/*018*/ IO_ERROR_LOG_PACKET     EntryData
/*048*/ }
        IO_ERROR_LOG_MESSAGE.
```

IO_ERROR_LOG_PACKET

```
typedef struct _IO_ERRDR_LOG_PACKET
        {
/*000*/ BYTE                    MajorFunctionCode.
/*001*/ BYTE                    RetryCount.
/*002*/ WORD                    DumpDataSize:
/*004*/ WORD                    NumberOfStrings.
/*006*/ WORD                    StringOffset:
/*008*/ WORD                    EventCategory:
/*00C*/ NTSTATUS                ErrorCode.
/*010*/ DWORD                   UniqueErrorValue.
/*014*/ NTSTATUS                FinalStatus.
/*018*/ DWORD                   SequenceNumber:
/*01C*/ DWORD                   IoControlCode:
/*020*/ LARGE_INTEGER           DeviceOffset
/*028*/ DWORD                   DumpData [1].
/*030*/ }
        IO_ERROR_LOG_PACKET.
```

IO_STATUS_BLOCK

```
typedef struct _IO_STATUS_BLOCK
        {
/*000*/ NTSTATUS     Status.
/*004*/ ULONG        Information:
/*008*/ }:
        IO_STATUS_BLOCK:
```

IO_TIMER

```
typedef struct _IO_TIMER
        {
/*000*/ SHORT                   Type:          // IO_TYPE_TIMER 0x09
/*002*/ WORD                    TimerState.    // 0 = остановлен. 1 = работает
/*004*/ LIST_ENTRY              TimerQueue.
/*00C*/ PIO_TIMER_ROUTINE       TimerRoutine:
/*010*/ PVOID                   Context.
/*014*/ PDEVICE_OBJECT          DeviceObject:
/*018*/ }
        IO_TIMER.
```

KAFFINITY

```
typedef DWORD KAFFINITY.
```

KAPC

```
typedef struct _KAPC
        {
/*000*/ SHORT                   Type:     // IO_TYPE_APC 0x12
/*002*/ SHORT                   Size:     // количество BYTE
/*004*/ DWORD                   Spare0:
/*008*/ struct _KTHREAD        *Thread:
/*00C*/ LIST_ENTRY              ApcListEntry:
/*014*/ PKKERNEL_ROUTINE        KernelRoutine:      // KiSuspendNop
/*018*/ PKRUNDOWN_ROUTINE       RundownRoutine:
/*01C*/ PKNORMAL_ROUTINE        NormalRoutine:      // KiSuspendThread
/*020*/ PVOID                   NormalContext:
/*024*/ PVOID                   SystemArgument1:
/*028*/ PVOID                   SystemArgument2:
/*02C*/ CHAR                    ApcStateIndex,
/*02D*/ KPROCESSOR_MODE         ApcMode:
/*02E*/ BOOLEAN                 Inserted:
/*030*/ }
        KAPC:
```

KAPC_STATE

```
typedef struct _KAPC_STATE
        {
/*000*/ LIST_ENTRY              ApcListHead [2]:
/*010*/ struct _KPROCESS       *Process:
/*014*/ BOOLEAN                 KernelApcInProgress:
/*015*/ BOOLEAN                 KernelApcPending:
/*016*/ BOOLEAN                 UserApcPending:
/*018*/ }
        KAPC_STATE:
```

KDEVICE_QUEUE

```
typedef struct _KDEVICE_QUEUE
        {
/*000*/ SHORT        Type: // IO_TYPE_DEVICE_QUEUE 0x14
/*002*/ SHORT        Size: // количество BYTE
/*004*/ LIST_ENTRY   DeviceListHead:
/*00C*/ KSPIN_LOCK   Lock:
/*010*/ BOOLEAN      Busy:
/*014*/ }
        KDEVICE_QUEUE:
```

KDEVICE_QUEUE_ENTRY

```
typedef struct _KDEVICE_QUEUE_ENTRY
        {
/*000*/ LIST_ENTRY   DeviceListEntry:
/*008*/ DWORD        SortKey:
/*00C*/ BOOLEAN      Inserted:
/*010*/ }
        KDEVICE_QUEUE_ENTRY:
```

KDPC

```
typedef struct _KDPC
        {
/*000*/ SHORT                   Type; // IO_TYPE_DPC 0x13
/*002*/ BYTE                    Number;
/*003*/ BYTE                    Importance;
/*004*/ LIST_ENTRY              DpcListEntry;
/*00C*/ PKDEFERRED_ROUTINE      DeferredRoutine;
/*010*/ PVOID                   DeferredContext;
/*014*/ PVOID                   SystemArgument1;
/*018*/ PVOID                   SystemArgument2;
/*01C*/ PDWORD_PTR              Lock;
/*020*/ }
        KDPC;
```

KEVENT

```
typedef struct _KEVENT
        {
/*000*/ DISPATCHER_HEADER Header;  // DISP_TYPE_*_EVENT
                                   // 0x00. 0x01
/*010*/ }
        KEVENT;
```

KEVENT_PAIR

```
typedef struct _KEVENT_PAIR
        {
/*000*/ SHORT        Type; // IO_TYPE_EVENT_PAIR 0x15
/*002*/ WORD         Size; // количество BYTE
/*004*/ KEVENT       Event1;
/*014*/ KEVENT       Event2;
/*024*/ }
        KEVENT_PAIR;
```

KGDTENTRY

```
typedef struct _KGDTENTRY
        {
/*000*/ WORD         LimitLow;
/*002*/ WORD         BaseLow;
/*004*/ DWORD        HighWord;
/*008*/ }
        KGDTENTRY;
```

KIDTENTRY

```
typedef struct _KIDTENTRY
        {
/*000*/ WORD Offset;
/*002*/ WORD Selector;
/*004*/ WORD Access;
/*006*/ WORD ExtendedOffset;
/*008*/ }
        KIDTENTRY;
```

KIRQL

```
typedef BYTE KIRQL;
```

KMUTANT, KMUTEX

```
typedef struct _KMUTANT
        {
/*000*/ DISPATCHER_HEADER  Header; // DISP_TYPE_MUTANT 0x02
/*010*/ LIST_ENTRY         MutantListEntry;
/*018*/ struct _KTHREAD    *OwnerThread;
/*01C*/ BOOLEAN            Abandoned;
/*01D*/ BYTE               ApcDisable;
/*020*/ }
        KMUTANT,           KMUTEX;
```

KPCR

```
// базовый адрес 0xFFDFF000

// область управления процессором
// (processor control region)
typedef struct _KPCR
        {
/*000*/ NT_TIB               NtTib;
/*01C*/ struct _KPCR         *SelfPcr;
/*020*/ PKPRCB               Prcb;
/*024*/ KIRQL                Irql;
/*028*/ DWORD                IRR;
/*02C*/ DWORD                IrrActive;
/*030*/ DWORD                IDR;
/*034*/ DWORD                Reserved2;
/*038*/ struct _KIDTENTRY    *IDT;
/*03C*/ struct _KGDTENTRY    *GDT;
/*040*/ struct _KTSS         *TSS;
/*044*/ WORD                 MajorVersion;
/*046*/ WORD                 MinorVersion;
/*048*/ KAFFINITY            SetMember.
/*04C*/ DWORD                StallScaleFactor;
/*050*/ BYTE                 DebugActive.
/*051*/ BYTE                 Number;
/*054*/ }
        KPCR;
```

KPRCB

```
// базовый адрес 0xFFDFF120

// блок управления процессором (processor control block)
typedef struct _KPRCB
        {
/*000*/ WORD                   MinorVersion;
/*002*/ WORD                   MajorVersion;
/*004*/ struct _KTHREAD        *CurrentThread;
/*008*/ struct _KTHREAD        *NextThread;
```

```
/*00C*/ struct _KTHREAD          *IdleThread,
/*010*/ CHAR                      Number;
/*011*/ CHAR                      Reserved;
/*012*/ WORD                      BuildType;
/*014*/ KAFFINITY                 SetMember;
/*018*/ struct _RESTART_BLOCK    *RestartBlock;
/*01C*/ }
        KPRCB;
```

KPROCESS

```
typedef struct _KPROCESS
        {
/*000*/ DISPATCHER_HEADER Header; // DO_TYPE_PROCESS (0x1B)
/*010*/ LIST_ENTRY              ProfileListHead.
/*018*/ DWORD                   DirectoryTableBase;
/*01C*/ DWORD                   PageTableBase.
/*020*/ KGDTENTRY               LdtDescriptor;
/*028*/ KIDTENTRY               Int21Descriptor;
/*030*/ WORD                    IopmOffset;
/*032*/ BYTE                    Iopl
/*033*/ BOOLEAN                 VdmFlag;
/*034*/ DWORD                   ActiveProcessors;
/*038*/ DWORD                   KernelTime:     // тактов
/*03C*/ DWORD                   UserTime.       // тактов
/*040*/ LIST_ENTRY              ReadyListHead;
/*048*/ LIST_ENTRY              SwapListEntry;  // KTHREAD.ThreadListEntry
/*050*/ LIST_ENTRY              ThreadListHead;
/*058*/ PVOID                   ProcessLock;
/*05C*/ KAFFINITY               Affinity;
/*060*/ WORD                    StackCount;
/*062*/ BYTE                    BasePriority;
/*063*/ BYTE                    ThreadQuantum;
/*064*/ BOOLEAN                 AutoAlignment;
/*065*/ BYTE                    State.
/*066*/ BYTE                    ThreadSeed.
/*067*/ BOOLEAN                 DisableBoost;
/*068*/ DWORD                   d068;
/*06C*/ }
        KPROCESS;
```

KPROCESSOR_MODE

```
typedef CHAR KPROCESSOR_MODE
```

KQUEUE

```
typedef struct _KQUEUE
        {
/*000*/ DISPATCHER_HEADER Header; // DISP_TYPE_QUEUE 0x04
/*010*/ LIST_ENTRY              EntryListHead;
/*018*/ DWORD                   CurrentCount;
/*01C*/ DWORD                   MaximumCount;
/*020*/ LIST_ENTRY              ThreadListHead;
/*028*/ }
        KQUEUE;
```

KSEMAPHORE

```
typedef struct _KSEMAPHORE
        {
/*000*/ DISPATCHER_HEADER      Header; // DISP_TYPE_SEMAPHORE 0x05
/*010*/ LONG                   Limit;
/*014*/ }
        KSEMAPHORE;
```

KTHREAD

```
typedef struct _KTHREAD
        {
/*000*/ DISPATCHER_HEADER          Header; // DO_TYPE_THREAD (0x6C)
/*010*/ LIST_ENTRY                 MutantListHead;
/*018*/ PVOID                      InitialStack.
/*01C*/ PVOID                      StackLimit;
/*020*/ struct _TEB                *Teb.
/*024*/ PVOID                      TlsArray;
/*028*/ PVOID                      KernelStack;
/*02C*/ BOOLEAN                    DebugActive.
/*02D*/ BYTE                       State  // THREAD_STATE_*
/*02E*/ BOOLEAN                    Alerted;
/*02F*/ BYTE                       bReserved01;
/*030*/ BYTE                       Iopl;
/*031*/ BYTE                       NpxState.
/*032*/ BYTE                       Saturation.
/*033*/ BYTE                       Priority;
/*034*/ KAPC_STATE                 ApcState;
/*04C*/ DWORD                      ContextSwitches.
/*050*/ DWORD                      WaitStatus;
/*054*/ BYTE                       WaitIrql.
/*055*/ BYTE                       WaitMode;
/*056*/ BYTE                       WaitNext
/*057*/ BYTE                       WaitReason;
/*058*/ PLIST_ENTRY                WaitBlockList;
/*05C*/ LIST_ENTRY                 WaitListEntry;
/*064*/ DWORD                      WaitTime.
/*068*/ BYTE                       BasePriority;
/*069*/ BYTE                       DecrementCount;
/*06A*/ BYTE                       PriorityDecrement.
/*06B*/ BYTE                       Quantum.
/*06C*/ KWAIT_BLOCK                WaitBlock [4];
/*0CC*/ DWORD                      LegoData.
/*0D0*/ DWORD                      KernelApcDisable;
/*0D4*/ KAFFINITY                  UserAffinity;
/*0D8*/ BOOLEAN                    SystemAffinityActive;
/*0D9*/ BYTE                       Pad [3].
/*0DC*/ PSERVICE_DESCRIPTOR_TABLE  pServiceDescriptorTable;
/*0E0*/ PVOID                      Queue;
/*0E4*/ PVOID                      ApcQueueLock;
/*0E8*/ KTIMER                     Timer;
/*110*/ LIST_ENTRY                 QueueListEntry;
/*118*/ KAFFINITY                  Affinity;
/*11C*/ BOOLEAN                    Preempted;
```

```
/*11D*/ BOOLEAN                 ProcessReadyQueue;
/*11E*/ BOOLEAN                 KernelStackResident;
/*11F*/ BYTE                    NextProcessor;
/*120*/ PVOID                   CallbackStack;
/*124*/ struct _WIN32_THREAD    *Win32Thread;
/*128*/ PVOID                   TrapFrame;
/*12C*/ PKAPC_STATE             ApcStatePointer;
/*130*/ PVOID                   p130.
/*134*/ BOOLEAN                 EnableStackSwap;
/*135*/ BOOLEAN                 LargeStack;
/*136*/ BYTE                    ResourceIndex;
/*137*/ KPROCESSOR_MODE         PreviousMode;
/*138*/ DWORD                   KernelTime;       // тактов
/*13C*/ DWORD                   UserTime.         // тактов
/*140*/ KAPC_STATE              SavedApcState;
/*157*/ BYTE                    bReserved02;
/*158*/ BOOLEAN                 Alertable;
/*159*/ BYTE                    ApcStateIndex;
/*15A*/ BOOLEAN                 ApcQueueable;
/*15B*/ BOOLEAN                 AutoAlignment;
/*15C*/ PVOID                   StackBase.
/*160*/ KAPC                    SuspendApc;
/*190*/ KSEMAPHORE              SuspendSemaphore;
/*1A4*/ LIST_ENTRY              ThreadListEntry;  // см. KPROCESS
/*1AC*/ BYTE                    FreezeCount;
/*1AD*/ BYTE                    SuspendCount;
/*1AE*/ BYTE                    IdealProcessor;
/*1AF*/ BOOLEAN                 DisableBoost;
/*1B0*/ }
        KTHREAD;
```

KTIMER

```
typedef struct _KTIMER
        {
/*000*/ DISPATCHER_HEADER Header; // DISP_TYPE_*_TIMER 0x08. 0x09
/*010*/ ULARGE_INTEGER          DueTime;
/*018*/ LIST_ENTRY              TimerListEntry;
/*020*/ struct _KDPC            *Dpc;
/*024*/ LONG                    Period;
/*028*/ }
        KTIMER;
```

KWAIT_BLOCK

```
typedef struct _KWAIT_BLOCK
        {
/*000*/ LIST_ENTRY              WaitListEntry;
/*008*/ struct _KTHREAD         *Thread;
/*00C*/ PVOID                   Object;
/*010*/ struct _KWAIT_BLOCK     *NextWaitBlock;
/*004*/ WORD                    WaitKey;
/*006*/ WORD                    WaitType;
/*01B*/ }
        KWAIT_BLOCK;
```

LARGE_INTEGER

```
typedef union _LARGE_INTEGER
        {
/*000*/    struct
             {
/*000*/       ULONG     LowPart;
/*004*/       LONG      HighPart;
/*00B*/       }.
/*000*/    LONGLONG      QuadPart;
/*008*/    }
           LARGE_INTEGER;
```

LIST_ENTRY

```
typedef struct _LIST_ENTRY
        {
/*000*/ struct _LIST_ENTRY *Flink;
/*004*/ struct _LIST_ENTRY *Blink;
/*008*/ }
        LIST_ENTRY.
```

MMSUPPORT

```
typedef struct _MMSUPPORT
        {
/*000*/ LARGE_INTEGER     LastTrimTime;
/*008*/ DWORD             LastTrimFaultCount.
/*00C*/ DWORD             PageFaultCount.
/*010*/ DWORD             PeakWorkingSetSize;
/*014*/ DWORD             WorkingSetSize.
/*018*/ DWORD             MinimumWorkingSetSize.
/*01C*/ DWORD             MaximumWorkingSetSize;
/*020*/ PVOID             VmWorkingSetList;
/*024*/ LIST_ENTRY        WorkingSetExpansionLinks.
/*02C*/ BOOLEAN           AllowWorkingSetAdjustment
/*02D*/ BOOLEAN           AddressSpaceBeingDeleted;
/*02E*/ BYTE              ForegroundSwitchCount
/*02F*/ BYTE              MemoryPriority.
/*030*/ }
        MMSUPPORT;
```

NT_TIB (Thread Information Block, блок информации потока)

```
typedef struct _NT_TIB // см. winnt h / ntddk.h
        {
/*000*/ struct _EXCEPTION_REGISTRATION_RECORD     *ExceptionList.
/*004*/ PVOID                                      StackBase;
/*008*/ PVOID                                      StackLimit;
/*00C*/ PVOID                                      SubSystemTib;
/*010*/ union
```

```
        {
/*010*/    PVOID          FiberData,
/*010*/    ULONG          Version;
        };
/*014*/ PVOID             ArbitraryUserPointer,
/*018*/ struct _NT_TIB    *Self;
/*01C*/ }
        NT_TIB.
```

NTSTATUS

```
typedef LONG NTSTATUS;
```

OBJECT_ATTRIBUTES

```
typedef struct _OBJECT_ATTRIBUTES
        {
/*000*/ DWORD                          Length. // 0x18
/*004*/ HANDLE                         RootDirectory.
/*008*/ PUNICODE_STRING                ObjectName.
/*00C*/ DWORD                          Attributes.
/*010*/ PSECURITY_DESCRIPTOR           SecurityDescriptor;
/*014*/ PSECURITY_QUALITY_OF_SERVICE   SecurityQualityOfService;
/*018*/ }
        OBJECT_ATTRIBUTES;
```

OBJECT_CREATE_INFO

```
typedef struct _OBJECT_CREATE_INFO
        {
/*000*/ DWORD                          Attributes. // OBJ_*
/*004*/ HANDLE                         RootDirectory;
/*008*/ DWORD                          Reserved;
/*00C*/ KPROCESSOR_MODE                AccessMode;
/*010*/ DWDRD                          PagedPoolCharge
/*014*/ DWORD                          NonPagedPoolCharge;
/*018*/ DWORD                          SecurityCharge,
/*01C*/ PSECURITY_DESCRIPTOR           SecurityDescriptor;
/*020*/ PSECURITY_QUALITY_OF_SERVICE   SecurityQualityOfService;
/*024*/ SECURITY_QUALITY_OF_SERVICE    SecurityQualityOfServiceBuffer.
/*030*/ }
        OBJECT_CREATE_INFO;
```

OBJECT_CREATOR_INFO

```
typedef struct _OBJECT_CREATOR_INFO
        {
/*000*/ LIST_ENTRY        ObjectList;       // OBJECT_CREATOR_INFO
/*008*/ HANDLE            UniqueProcessId;
/*00C*/ WORD              Reserved1;
/*00E*/ WORD              Reserved2;
/*010*/ }
        OBJECT_CREATOR_INFO;
```

OBJECT_DIRECTORY

```
typedef struct _OBJECT_DIRECTORY
        {
/*000*/  POBJECT_DIRECTORY_ENTRY     HashTable [OBJECT_HASH_TABLE_SIZE];
/*094*/  POBJECT_DIRECTORY_ENTRY     CurrentEntry;
/*098*/  BOOLEAN                     CurrentEntryValid;
/*099*/  BYTE                        Reserved1;
/*09A*/  WORD                        Reserved2;
/*09C*/  DWORD                       Reserved3,
/*0A0*/  }
        OBJECT_DIRECTORY;
```

OBJECT_DIRECTORY_ENTRY

```
typedef struct _OBJECT_DIRECTORY_ENTRY
        {
/*000*/  struct _OBJECT_DIRECTORY_ENTRY    *NextEntry;
/*004*/  POBJECT                           Object;
/*008*/  }
        OBJECT_DIRECTORY_ENTRY.
```

OBJECT_HANDLE_DB

```
typedef struct _OBJECT_HANDLE_DB
        {
/*000*/  union
            {
/*000*/      struct _EPROCESS                  *Process,
/*000*/      struct _OBJECT_HANDLE_DB_LIST     *HandleOBList;
/*004*/      };
/*004*/  DWORD HandleCount;
/*008*/  }
        OBJECT_HANDLE_DB.
```

OBJECT_HANDLE_DB_LIST

```
typedef struct _OBJECT_HANDLE_DB_LIST
        {
/*000*/  DWORD                 Count,
/*004*/  OBJECT_HANDLE_DB      Entries [],
/*???*/  }
        OBJECT_HANOLE_DB_LIST;
```

OBJECT_HANDLE_INFORMATION

```
typedef struct _OBJECT_HANDLE_INFORMATION    // см. HANDLE_ENTRY
        {
/*000*/  DWORD            HandleAttributes;  // см. HANDLE_ATTRIBUTE_MASK
/*004*/  ACCESS_MASK      GrantedAccess,
/*008*/  }
        OBJECT_HANDLE_INFORMATION.
```

OBJECT_HEADER

```
typedef struct _OBJECT_HEADER
        {
/*000*/ DWORD            PointerCount;       // количество ссылок
/*004*/ DWORD            HandleCount;        // количество открытых дескрипторов
/*008*/ PDBJECT_TYPE     ObjectType;
/*00C*/ BYTE             NameOffset,         // -> DBJECT_NAME
/*00D*/ BYTE             HandleDBOffset;     // -> OBJECT_HANDLE_DB
/*00E*/ BYTE             QuotaChargesOffset; // -> OBJECT_QUDTA_CHARGES
/*00F*/ BYTE             ObjectFlags;        // OB_FLAG_*
/*010*/ union
            {   // OB_FLAG_CREATE_INFO ? ObjectCreateInfo , QuotaBlock
/*010*/     PQUOTA_BLOCK          QuotaBlock,
/*010*/     POBJECT_CREATE_INFO   ObjectCreateInfo;
/*014*/     },
/*014*/ PSECURITY_DESCRIPTOR      SecurityDescriptor,
/*018*/ }
        DBJECT_HEADER,
```

OBJECT_NAME

```
typedef struct _OBJECT_NAME
        {
/*000*/ POBJECT_DIRECTORY     Directory,
/*004*/ UNICODE_STRING        Name;
/*00C*/ DWORD                 Reserved,
/*010*/ }
        OBJECT_NAME;
```

OBJECT_NAME_INFORMATION

```
typedef struct _OBJECT_NAME_INFORMATION
        {
/*000*/ UNICODE_STRING   Name,     // указывает на Buffer[]
/*00B*/ WORD             Buffer [],
/*???*/ }
        OBJECT_NAME_INFORMATION,
```

OBJECT_QUOTA_CHARGES

```
typedef struct _OBJECT_QUOTA_CHARGES
        {
/*000*/ DWORD PagedPoolCharge;
/*004*/ DWORD NonPagedPoolCharge;
/*008*/ DWORD SecurityCharge,
/*00C*/ DWORD Reserved;
/*010*/ }
        OBJECT_QUOTA_CHARGES,
```

OBJECT_TYPE

```
typedef struct _OBJECT_TYPE
        {
/*000*/ ERESOURCE                Lock;
/*038*/ LIST_ENTRY               ObjectListHead, // OBJECT_CREATOR_INFO
/*040*/ UNICODE_STRING           ObjectTypeName, // см  выше
```

```
/*048*/ union
        {
/*048*/   PVOID DefaultObject;    // DbpDefaultObject
/*048*/   DWORD Code;             // File  5C, WaitablePort: A0
        };
/*04C*/ DWORD                    ObjectTypeIndex; // OB_TYPE_INDEX_*
/*050*/ DWORD                    ObjectCount;
/*054*/ DWORD                    HandleCount;
/*058*/ DWORD                    PeakObjectCount;
/*05C*/ DWORD                    PeakHandleCount;
/*060*/ OBJECT_TYPE_INITIALIZER  ObjectTypeInitializer;
/*0AC*/ DWORD                    ObjectTypeTag; // OB_TYPE_TAG_*
/*0B0*/ }
        OBJECT_TYPE;
```

OBJECT_TYPE_ARRAY

```
typedef struct _OBJECT_TYPE_ARRAY
        {
/*000*/ DWORD                    ObjectCount;
/*004*/ POBJECT_CREATOR_INFO     ObjectList [];
/*???*/ }
        OBJECT_TYPE_ARRAY;
```

OBJECT_TYPE_INFO

```
typedef struct _OBJECT_TYPE_INFO
        {
/*000*/ UNICODE_STRING   ObjectTypeName; // указывает на Buffer[]
/*008*/ DWORD            ObjectCount;
/*00C*/ DWORD            HandleCount;
/*010*/ DWORD            Reserved1 [4];
/*020*/ DWORD            PeakObjectCount;
/*024*/ DWORD            PeakHandleCount;
/*028*/ DWORD            Reserved2 [4];
/*038*/ DWORD            InvalidAttributes;
/*03C*/ GENERIC_MAPPING  GenericMapping;
/*04C*/ ACCESS_MASK      ValidAccessMask;
/*050*/ BOOLEAN          SecurityRequired;
/*051*/ BOOLEAN          MaintainHandleCount;
/*052*/ WORD             Reserved3;
/*054*/ BOOL             PagedPool,
/*05B*/ DWORD            DefaultPagedPoolCharge;
/*05C*/ DWORD            DefaultNonPagedPoolCharge;
/*060*/ WORD             Buffer [];
/*???*/ }
        OBJECT_TYPE_INFO;
```

OBJECT_TYPE_INITIALIZER

```
typedef struct _OBJECT_TYPE_INITIALIZER
        {
/*000*/ WORD                Length;           //0x004C  OBJECT_TYPE.DefaultObject
/*002*/ BOOLEAN             UseDefaultObject;
/*003*/ BOOLEAN             Reserved1;
```

```
/*004*/ DWORD                 InvalidAttributes;
/*008*/ GENERIC_MAPPING       GenericMapping.
/*018*/ ACCESS_MASK           ValidAccessMask;
/*01C*/ BOOLEAN               SecurityRequired;
/*01D*/ BOOLEAN               MaintainHandleCount:    // OBJECT_HANDLE_OB
/*01E*/ BOOLEAN               MaintainTypeList:       // OBJECT_CREATOR_INFO
/*01F*/ BYTE                  Reserved2;
/*020*/ BOOL                  PagedPool;
/*024*/ DWORD                 DefaultPagedPoolCharge;
/*028*/ DWORD                 DefaultNonPagedPoolCharge;
/*02C*/ NTPROC                DumpProcedure;
/*030*/ NTPROC                OpenProcedure;
/*034*/ NTPROC                CloseProcedure;
/*038*/ NTPROC                DeleteProcedure;
/*03C*/ NTPROC_VOID           ParseProcedure;         // SeDefaultObjectMethod
/*040*/ NTPROC_VOID           SecurityProcedure;
/*044*/ NTPROC_VOID           QueryNameProcedure;
/*048*/ NTPROC_BOOLEAN        OkayToCloseProcedure;
/*04C*/ }
        OBJECT_TYPE_INITIALIZER;
```

OEM_STRING

```
typedef STRING OEM_STRING;
```

OWNER_ENTRY

```
typedef struct _OWNER_ENTRY
        {
/*000*/ ERESOURCE_THREAD OwnerThread;
/*004*/ union
            {
/*004*/      LONG     OwnerCount;
/*004*/      DWORD    TableSize;
/*008*/      };
/*008*/ }
        OWNER_ENTRY;
```

PEB (Process Environment Block, блок переменных окружения процесса)

```
// расположен по адресу 0x7FFDF000

typedef struct _PEB
        {
/*000*/ BOOLEAN               InheritedAddressSpace;
/*001*/ BOOLEAN               ReadImageFileExecOptions;
/*002*/ BOOLEAN               BeingDebugged;
/*003*/ BYTE                  b003;
/*004*/ DWORD                 d004;
/*008*/ PVOID                 SectionBaseAddress;
/*00C*/ PPROCESS_MODULE_INFO  ProcessModuleInfo;
/*010*/ PPROCESS_PARAMETERS   ProcessParameters;
/*014*/ DWORD                 SubSystemData;
```

```
/*018*/ HANDLE              ProcessHeap;
/*01C*/ PCRITICAL_SECTION   FastPebLock;
/*020*/ PVOID               AcquireFastPebLock;  // функция
/*024*/ PVOID               ReleaseFastPebLock;  // функция
/*028*/ DWORD               d028.
/*02C*/ PPVOID              User32Dispatch;      // функция
/*030*/ DWORD               d030.
/*034*/ DWORD               d034.
/*038*/ DWORD               d038
/*03C*/ DWORD               TlsBitMapSize;       // количество бит
/*040*/ PRTL_BITMAP         TlsBitMap;           // ntdll!TlsBitMap
/*044*/ DWORD               TlsBitMapData [2];   // 64 бита
/*04C*/ PVOID               p04C.
/*050*/ PVOID               p050.
/*054*/ PTEXT_INFO          TextInfo.
/*058*/ PVOID               InitAnsiCodePageData;
/*05C*/ PVOID               InitOemCodePageData;
/*060*/ PVOID               InitUnicodeCaseTableData;
/*064*/ DWORD               KeNumberProcessors.
/*068*/ DWORD               NtGlobalFlag.
/*06C*/ DWORD               d6C;
/*070*/ LARGE_INTEGER       MmCriticalSectionTimeout;
/*078*/ DWORD               MmHeapSegmentReserve;
/*07C*/ DWORD               MmHeapSegmentCommit;
/*080*/ DWORD               MmHeapDeCommitTotalFreeThreshold;
/*084*/ DWORD               MmHeapDeCommitFreeBlockThreshold;
/*088*/ DWORD               NumberOfHeaps.
/*08C*/ DWORD               AvailableHeaps.    // 16. *2 если израсходованы
/*090*/ PHANDLE             ProcessHeapsListBuffer;
/*094*/ DWORD               d094;
/*098*/ DWORD               d098.
/*09C*/ DWORD               d09C.
/*0A0*/ PCRITICAL_SECTION   LoaderLock;
/*0A4*/ DWORD               NtMajorVersion;
/*0A8*/ DWORD               NtMinorVersion;
/*0AC*/ WORD                NtBuildNumber;
/*0AE*/ WORD                CmNtCSDVersion;
/*0B0*/ DWORD               PlatformId.
/*0B4*/ DWORD               Subsystem.
/*0B8*/ DWORD               MajorSubsystemVersion;
/*0BC*/ DWORD               MinorSubsystemVersion;
/*0C0*/ KAFFINITY           AffinityMask;
/*0C4*/ DWORD               ad0C4 [35].
/*150*/ PVOID               p150;
/*154*/ DWORD               ad154 [32];
/*1D4*/ HANDLE              Win32WindowStation;
/*1D8*/ DWORD               d1D8.
/*1DC*/ DWORD               d1DC;
/*1E0*/ PWORD               CSDVersion;
/*1E4*/ DWORD               d1E4;
/*1E8*/ }
        PEB;
```

PHYSICAL_ADDRESS

```
typedef LARGE_INTEGER PHYSICAL_ADDRESS;
```

PROCESS_PARAMETERS

```
typedef struct _PROCESS_PARAMETERS
        {
/*000*/ DWORD            Allocated;
/*004*/ DWORD            Size;
/*008*/ DWORD            Flags;    // бит 0  все указатели нормализованы
/*00C*/ DWORD            Reserved1;
/*010*/ LONG             Console;
/*014*/ DWORD            ProcessGroup;
/*018*/ HANDLE           StdInput;
/*01C*/ HANDLE           StdOutput;
/*020*/ HANDLE           StdError;
/*024*/ UNICODE_STRING   WorkingDirectoryName;
/*02C*/ HANDLE           WorkingDirectoryHandle;
/*030*/ UNICODE_STRING   SearchPath;
/*038*/ UNICODE_STRING   ImagePath;
/*040*/ UNICODE_STRING   CommandLine;
/*048*/ PWORD            Environment;
/*04C*/ DWORD            X,
/*050*/ DWORD            Y
/*054*/ DWORD            XSize;
/*058*/ DWORD            YSize;
/*05C*/ DWORD            XCountChars;
/*060*/ DWORD            YCountChars;
/*064*/ DWORD            FillAttribute;
/*068*/ DWORD            Flags2;
/*06C*/ WORD             ShowWindow;
/*06E*/ WORD             Reserved2;
/*070*/ UNICODE_STRING   Title;
/*078*/ UNICODE_STRING   Desktop;
/*080*/ UNICODE_STRING   Reserved3,
/*088*/ UNICODE_STRING   Reserved4;
/*090*/ }
        PROCESS_PARAMETERS;
```

QUOTA_BLOCK

```
typedef struct _QUOTA_BLOCK
        {
/*000*/ DWORD Flags,
/*004*/ DWORD ChargeCount,
/*008*/ DWORD PeakPoolUsage    [2]; // невыгружаемый пул, выгружаемый пул
/*010*/ DWORD PoolUsage        [2]; // невыгружаемый пул, выгружаемый пул
/*018*/ DWORD PoolQuota        [2]; // невыгружаемый пул, выгружаемый пул
/*020*/ }
        QUOTA_BLOCK;
```

RTL_BITMAP

```
typedef struct _RTL_BITMAP
        {
/*000*/ DWORD        SizeOfBitMap;
/*004*/ PDWORD       Buffer;
/*008*/ }
        RTL_BITMAP;
```

RTL_CRITICAL_SECTION_DEBUG

```
#define RTL_CRITSECT_TYPE 0
#define RTL_RESOURCE_TYPE 1

typedef struct _RTL_CRITICAL_SECTION_DEBUG
        {
/*000*/ WORD                                Type;
/*002*/ WORD                                CreatorBackTraceIndex;
/*004*/ struct _RTL_CRITICAL_SECTION       *CriticalSection;
/*008*/ LIST_ENTRY                          ProcessLocksList;
/*010*/ DWORD                               EntryCount;
/*014*/ DWORD                               ContentionCount;
/*018*/ DWORD                               Spare [2];
/*020*/ }
        RTL_CRITICAL_SECTION_DEBUG;
```

SECTION_OBJECT_POINTERS

```
typedef struct _SECTION_OBJECT_POINTERS
        {
/*000*/ PVOID DataSectionObject;
/*004*/ PVOID SharedCacheMap;
/*008*/ PVOID ImageSectionObject;
/*00C*/ }
        SECTION_OBJECT_POINTERS;
```

SECURITY_DESCRIPTOR

```
typedef struct _SECURITY_DESCRIPTOR
        {
/*000*/ BYTE                               Revision;
/*001*/ BYTE                               Sbz1;
/*002*/ SECURITY_DESCRIPTOR_CONTROL        Control;
/*004*/ PSID                               Owner;
/*008*/ PSID                               Group;
/*00C*/ PACL                               Sacl;
/*010*/ PACL                               Oacl;
/*014*/ }
        SECURITY_DESCRIPTOR;
```

SECURITY_DESCRIPTOR_CONTROL

```
typedef WORD SECURITY_DESCRIPTOR_CONTROL;
```

SERVICE_DESCRIPTOR_TABLE

```
typedef struct _SERVICE_DESCRIPTOR_TABLE
        {
/*000*/ SYSTEM_SERVICE_TABLE ntoskrnl;      // ntoskrnl exe (native api)
/*010*/ SYSTEM_SERVICE_TABLE win32k;        // win32k sys (gdi/user)
/*020*/ SYSTEM_SERVICE_TABLE Table3;        // не используется
/*030*/ SYSTEM_SERVICE_TABLE Table4;        // не используется
/*040*/ }
        SERVICE_DESCRIPTOR_TABLE;
```

STRING

```
typedef struct _STRING
        {
/*000*/ WORD    Length.
/*002*/ WORD    MaximumLength.
/*004*/ PBYTE   Buffer.
/*008*/ }
        STRING.
```

SYSTEM_SERVICE_TABLE

```
typedef struct _SYSTEM_SERVICE_TABLE
        {
/*000*/ PNTPROC      ServiceTable.       // массив точек входа
/*004*/ POWORD       CounterTable.       // массив счетчиков использования
/*008*/ DWORD        ServiceLimit;       // число записей в таблицы
/*00C*/ PBYTE        ArgumentTable;      // массив счетчиков байт
/*010*/ }
        SYSTEM_SERVICE_TABLE;
```

TEB (Thread Environment Block, блок переменных окружения потока)

```
typedef struct _TEB
        {
/*000*/ NT_TIB           Tib;
/*01C*/ PVOID            EnvironmentPointer;
/*020*/ CLIENT_ID        Cid;
/*028*/ HANDLE           RpcHandle;
/*02C*/ PPVOID           ThreadLocalStorage.
/*030*/ PPEB             Peb;
/*034*/ DWORD            LastErrorValue;
/*038*/ }
        TEB;
```

TIME_FIELDS

```
typedef struct _TIME_FIELDS
        {
/*000*/ SHORT Year;
/*002*/ SHORT Month;
/*004*/ SHORT Day.
/*006*/ SHORT Hour.
/*008*/ SHORT Minute;
/*00A*/ SHORT Second;
/*00C*/ SHORT Milliseconds.
/*00E*/ SHORT Weekday;    // 0 = воскресенье
/*010*/ }
        TIME_FIELDS;
```

ULARGE_INTEGER

```
typedef union _ULARGE_INTEGER
        {
```

```
/*000*/ struct
        {
/*000*/   ULONG LowPart;
/*004*/   ULONG HighPart;
/*008*/   }.
/*000*/ ULONGLONG QuadPart;
/*008*/ }
        ULARGE_INTEGER;
```

UNICODE_STRING

```
typedef struct _UNICODE_STRING
        {
/*000*/ USHORT       Length;
/*002*/ USHORT       MaximumLength;
/*004*/ PWSTR        Buffer;
/*008*/ }
        UNICODE_STRING;
```

VPB (Volume Parameter Block, блок характеристик тома)

```
// volume parameter block, блок характеристик тома
typedef struct _VPB // volume parameter block
        {
/*000*/ SHORT                    Type;  // IO_TYPE_VPB 0x0A
/*002*/ SHORT                    Size;  // число BYTE
/*004*/ WORD                     Flags;
/*006*/ WORD                     VolumeLabelLength; // байт (без завершающего нуля)
/*008*/ struct _DEVICE_OBJECT   *DeviceObject;
/*00C*/ struct _DEVICE_OBJECT   *RealDevice;
/*010*/ DWORD                    SerialNumber;
/*014*/ DWORD                    ReferenceCount;
/*018*/ WORD                     VolumeLabel [MAXIMUM_VOLUME_LABEL];
/*058*/ }
        VPB;
```

WAIT_CONTEXT_BLOCK

```
typedef struct _WAIT_CONTEXT_BLOCK
        {
/*000*/ KDEVICE_QUEUE_ENTRY        WaitQueueEntry;
/*010*/ PDRIVER_CONTROL            DeviceRoutine;
/*014*/ PVOID                      DeviceContext;
/*018*/ DWORD                      NumberOfMapRegisters;
/*01C*/ PVOID                      DeviceObject;
/*020*/ PVOID                      CurrentIrp;
/*024*/ PKDPC                      BufferChainingDpc;
/*028*/ }
        WAIT_CONTEXT_BLOCK;
```

Алфавитный указатель

E

F

G

H

I

K

L

M

N

O

P

T

U

V

W

X

Z

А

Б

В

Д

З

И

К

Л

М

Н

Т

У

Ф

Ш

Э

Я