ВАЛЕРИЯ КОМИССАРОВА

bhv®

ПРОГРАММИРОВАНИЕ ДРАЙВЕРОВ для Windows

Архитектуры WDM и WDF

Драйверы для всей линейки операционных систем Windows NT, включая Windows Vista

Драйверы для многопроцессорных систем

Драйверы для видеокарты, USB-камеры и др.

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Валерия Комиссарова

ПРОГРАММИРОВАНИЕ ДРАЙВЕРОВ для Windows

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.1
К63

Комиссарова В.

К63 Программирование драйверов для Windows. — СПб.: БХВ-Петербург, 2007. — 256 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0023-4

Книга представляет собой практическое руководство по программированию драйверов для всей линейки операционных систем Windows NT, включая новую ОС Windows Vista. Разбираются важнейшие драйверные архитектуры — традиционная WDM и новая WDF. Излагаются основы теории программирования драйверов для многопроцессорных систем. Показано, как создать простейший драйвер, а также приведены практические примеры написания сложных драйверов для принтера, монитора, видеокарты и USB-камеры.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 05.03.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 20,64.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0023-6

Оглавление

Очень, очень хорошему человеку — О. Ф. Литвину

Предисловие

Драйверная концепция — неотъемлемая часть современных операционных систем. Эта концепция — основа взаимодействия системы (пользователя) с какими бы то ни было устройствами (системными/периферийными, реальными/виртуальными и т. д.). К сожалению, даже системные программисты (не говоря уже о прикладных программистах или, вовсе, о рядовых пользователях) далеко не всегда имеют какое-либо представление об этой концепции, о принципах ее работы, о программировании с использованием этой концепции. А, как известно, системное программирование — ключ к пониманию основ IT. Тем более, такой его раздел, как написание драйверов. Поэтому необходимость качественного изучения его — очевидна.

Написание драйверов — достаточно сложная, но, тем не менее, очень интересная и, что немаловажно, актуальная отрасль программирования. Знание особенностей технологий написания драйверов открывает огромное количество возможностей — написание драйверов для устройств, уже не поддерживаемых производителем, для устройств, драйверы к которым еще не написаны, исправление ошибок в драйверах, написание драйверов к различным промышленным устройствам... Список можно продолжать долго.

Большую помощь в деле освоения какого бы то ни было раздела программирования оказывает соответствующая литература. Но, к сожалению, на российском рынке компьютерной литературы остро ощущается нехватка изданий по написанию драйверов для различных операционных систем и платформ (особенно в последнее время). Кроме трех-четырех книг (причем далеко не все из них хотя бы приемлемого качества) — больше ничего нет. В результате этого наблюдения и возникла идея написания подобной книги.

Тема написания драйверов настолько огромна, что в такой маленькой по объему книге невозможно рассказать и половины того, что хотелось бы, и многие темы описаны лишь обзорно. Во-первых, не стоит забывать об операционных системах, отличных от семейства операционных систем Windows, использующихся повсеместно (самая популярная из которых, пожалуй, Linux), написание драйверов для которых осталось за рамками книги. Во-вторых, следует помнить о большом количестве устройств, написание драйверов для которых — очень трудное дело, со своей спецификой, требующее нетривиальных знаний и т. д. Все эти темы мы оставим в надежде на появление в будущем хорошей литературы по вышеуказанным и многим

другим предметам, входящим в обширнейшую область знаний под названием "программирование драйверов".

Актуальность темы программирования драйверов не уменьшается в течение уже долгого времени. Меняются лишь какие-либо драйверные модели (как, например, в случае с WDM на WDF, описанном в этой книге), но смены концепции драйверов как таковой не предвидится еще очень долго. Этому есть свои причины. Концепция драйверов до сих пор жива далеко не только потому, что IT-индустрия просто привыкла к ней. Драйверная концепция обладает рядом неоспоримых преимуществ, которые позволяют ей оставаться "на плаву". Изменение драйверной концепции "тянет" за собой изменение большого количества компонентов (таких как архитектура существующих операционных систем, например), тесно с ней взаимосвязанных (и наоборот). Пока что, повторяю, это не предвидится.

Структура книги

- **Глава 1. Основные понятия разработки драйверов.**

  В данной, вводной главе разъясняются основные понятия и концепции, с которыми сталкивается любой программист драйверов. Прочтение этой главы даст вам возможность уже без особых трудностей воспринимать последующий материал, изобилующий специфическими терминами и понятиями. Также в этой главе дается обзор самых главных и популярных (что важно, заслуженно) инструментов для написания драйверов.

- **Глава 2. Архитектура Windows.**

  В этой главе дается краткий обзор архитектуры операционной системы Windows. Хорошо знать и понимать архитектуру операционной системы, код для которой пишет системный программист, ему совершенно необходимо.

- **Глава 3. Архитектура WDM.**

  Здесь приводится подробное описание драйверной архитектуры WDM, которая долгое время являлась главной технологией и концепцией написания драйверов для ОС Windows. Вам необходимо разбираться в этой технологии, чтобы писать хорошие драйверы, грамотно использующие концепцию, с помощью которой они создаются, для любых операционных систем Windows, за исключением самых последних, использующих новую драйверную модель WDF.

- **Глава 4. Программирование в режиме ядра.**

  Процесс программирования в режиме ядра имеет очень существенные отличия от такового в пользовательском режиме. Фактически вы должны заново научиться программировать. Это не громкая фраза — в режиме

ядра свои "законы" программирования, свое API и т. д. Написание драйверов режима ядра невозможно без знания и понимания этих различий — главные из которых и описываются в этой главе.

- **Глава 5. Структура драйвера.**

  В этой главе показана и объяснена общая структура кода любого драйвера — как простого, так и сложного. Знание и понимание этой структуры помогает писать хорошо оформленный, удобный для чтения, исправления и сопровождения, эффективный код драйвера.

- **Глава 6. Простейший драйвер для Windows.**

  В этой главе подробно описывается весь процесс создания простейшего драйвера с минимальными функциями для ОС Windows NT — от написания самого кода, его компиляция и инсталляции до отладки драйвера. Глава даст вам (ну или, во всяком случае, попытается это сделать) все необходимые знания для осуществления этого процесса и немного больше того.

- **Глава 7. Сложные драйверы для Windows.**

  В этой главе подробно описан процесс написания настоящих сложных драйверов с большим количеством функций для определенных устройств — принтера, монитора и видеокарты, фильтр-драйвера камеры. Написанию драйверов для каждого из вышеперечисленных типов устройств можно посвятить отдельную книгу (пусть и не очень большую). В противовес этому — всего одна глава. Ее задача — объяснить главные принципы, концепции написания "полноценных" драйверов, выполняющих сложную работу с устройством, рассказать о характерных приемах, используемых при этом, дать представление о спектре знаний, необходимых для успешной работы в этой области, и т. д. и т. п.

- **Глава 8. Мультипроцессорная парадигма программирования.**

  Так как в этой книге, помимо всего прочего, рассказывается и о написании многопоточных драйверов, то я считаю необходимым, кроме того, рассказать и объяснить, что такое многопроцессорные системы вообще, какова их архитектура и особенности (как высоко-, так и низкоуровневые), в чем суть новой многопоточной парадигмы программирования и т. д. Эта вводная информация абсолютно необходима для полноценного понимания основ и принципов многопроцессорных систем и написания профессиональных драйверов для них.

- **Глава 9. Написание 64-битных драйверов и драйверов для многопроцессорных систем.**

  В этой главе рассказывается о написании многопоточных и 64-битных драйверов.

- **Глава 10. Новая операционная система Microsoft — Windows Vista.**

  Прежде чем приступать к рассказу о написании драйверов под новейшую ОС Windows Vista от компании Microsoft, необходимо опять-таки тщательно разобраться в самой системе — в ее новых возможностях, особенностях и т. д. В этом вам поможет материал данной главы. Мы рассмотрим сначала общую перспективу ОС Vista, затем перейдем к изучению более низкоуровневых ее особенностей.

- **Глава 11. Windows Driver Foundation (WDF).**

  Вместе с новой ОС Vista компания Microsoft, соответственно, выпустила и новую драйверную модель WDF. В этой главе подробно рассказывается об этой новой технологии, без знания которой нельзя писать качественные драйверы под Vista. Вы получите все необходимые для написания драйверов с использованием этой модели знания.

- **Глава 12. Написание драйверов в Vista — KMDF.**

  В этой главе рассказывается о написании драйверов режима ядра в Vista с использованием KMDF — Kernel-Mode Driver Framework (среда для написания драйверов режима ядра). Концепция "от простого — к сложному", важные детали, примеры кода — все, что нужно для того, чтобы свободно чувствовать себя в новой ОС при написании драйверов режима ядра, уметь легко разбираться в новых знаниях и получать их.

  Задача этой и следующей глав — не "изобрести велосипед", а попытаться как-то "скрасить" недостатки имеющейся документации по WDF, которая, на мой взгляд, в настоящий момент является чрезвычайно трудно понятной начинающему программисту. Сделана попытка объяснить неясные моменты в имеющейся документации и исходных кодах и более-менее систематизировать имеющиеся данные.

  Отмечу, что задача эта чрезвычайно трудная — документация постоянно развивается, поэтому описывать какие-то мелкие и сложные детали реализации пока, к сожалению, представляется мало возможным — поэтому сделана такая выборка информации, которая является основной; ее изменения достаточно маловероятны.

- **Глава 13. Написание драйверов в Vista — UMDF.**

  В этой главе рассказывается о написании драйверов пользовательского режима в Vista с помощью UMDF — User-Mode Driver Framework (среда для написания драйверов пользовательского режима). То же, что в предыдущей главе — но для драйверов пользовательского режима. Те же замечания, что и к предыдущей главе.

- Приложения.
  - **Приложение 1. Краткий словарь терминов.**
  - **Приложение 2. Полезные исходные коды из DDK.**

- **Приложение 3. Полезные исходные коды из KMDF.**
- **Приложение 4. Полезные исходные коды из UMDF.**

В приложениях к книге размещены: во-первых, справочная информация (об архитектуре WDM Streaming — знать ее нужно для успешного и осознанного написания драйверов для устройств, работающих с потоками, как, например, камеры) и краткий словарь самых необходимых терминов; а во-вторых, специально подобранные и наиболее полезные (в рамках охвата тем данной книги) исходные коды из DDK, UMDF и KMDF.

❒ **Список полезной литературы.**

Здесь приведен перечень полезной литературы.

Кому адресована эта книга

Данная книга предназначена для всех, кто хочет ознакомиться с азами написания драйверов. В ней освещаются вопросы написания драйверов как под Windows серии NT, так и под новейшую версию Windows — Vista. Текст книги построен по принципу "от простого — к сложному", поэтому вероятность возникновения проблем при написании более сложных драйверов была сведена к минимуму. В конце книги размещена самая необходимая справочная информация, теоретические главы (рассказывающие, например, об архитектуре WDM) чередуются с практическими примерами, — закрепляющими теорию. Все это, вместе взятое, способствует улучшению восприятия материала книги, а также делает ее интересной как для начинающих, так и для искушенных в деле написания драйверов читателей. Тем не менее, минимальные требования к читателю все же есть — знание языка C и хотя бы минимальные знания в области системного программирования (т. е. "начинающий" — предполагается только в деле написания драйверов, а не в программировании вообще).

Отмечу, что книга в наибольшей степени имеет практический характер; поэтому во всех главах предпочтение отдается практическим навыкам (пусть даже пока минимальным), а не теоретическим обоснованиям.

Об авторе

Комиссарова Валерия — обладатель сертификатов MCP, MCSD .NET. Имеет публикации в журналах "Хакер" и "IT-Спец" (бывший "Хакер-Спец"). Автор статей на сайтах **www.xakep.ru** и **www.securitylab.ru**.

Глава 1

Основные понятия разработки драйверов

В этой главе читатель получит минимум информации, необходимой для успешного понимания и изучения дальнейших глав этой книги.

Здесь мы рассмотрим основные понятия и термины, используемые в программировании драйверов, и инструменты, которые чаще всего применяются для написания драйверов.

1.1. Общие понятия

Изучение программирования драйверов — так же, как и изучение чего бы то ни было — нужно начинать с изучения теоретических основ. Так и поступим.

Прежде всего — базовые понятия. Итак, что такое драйвер? *Драйвер* — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические. Но это уже подводит нас к вопросу о том, какие вообще бывают драйверы (и, соответственно, для каких устройств), а об этом мы поговорим позднее.

С момента своего появления до сегодняшнего дня драйвер беспрерывно эволюционировал, и процесс этот до сих пор не закончился. Один из моментов эволюции драйвера — это эволюция концепции драйвера, как легко заменяемой части операционной системы. Как отдельный и довольно независимый модуль, драйвер сформировался не сразу. Да и сейчас многие драйверы практически неотделимы от операционной системы. Во многих случаях это приводит к необходимости переустановки системы (ОС

Windows) или пересборки ее (ядра) (в UNIX-системах). Такое же различие есть и между ветками операционной системы Windows: Windows 9*x* и Windows NT. В первом случае процесс работы с драйверами происходит (практически всегда) как с отдельными "кирпичиками", а во втором дела обстоят намного хуже (множество (если не большинство) драйверов "вшито" в ядро).

Список основных общих концепций драйверов в Windows- и UNIX-системах выглядит так:

- ❒ способ работы с драйверами как файлами;
- ❒ драйвер, как легко заменяемая часть ОС (учитывая сказанное выше);
- ❒ существование режима ядра.

Объясню подробнее первый пункт. Способ работы с драйверами как файлами означает, что функции, используемые при взаимодействии с файлами, практически идентичны таковым при взаимодействии с драйверами (имеется в виду лексически): `open`, `close`, `read` и т. д. О режиме ядра я расскажу позднее.

И напоследок стоит отметить (добавить к нашему списку) идентичность механизма IOCTL (Input/Output Control Code, код управления вводом/выводом) — запросов.

Теперь рассмотрим классификацию типов драйверов (замечу, довольно условную) для ОС Windows NT:

- ❒ драйверы пользовательского режима (User-Mode Drivers):
  - драйверы виртуальных устройств (Virtual Device Drivers, VDD) — используются для поддержки программ MS-DOS;
  - драйверы принтеров (Printer Drivers);
- ❒ драйверы режима ядра (Kernel-Mode Drivers):
  - драйверы файловой системы (File System Drivers) — осуществляют ввод/вывод на локальные и сетевые диски;
  - унаследованные драйверы (Legacy Drivers) — написаны для предыдущих версий Windows NT;
  - драйверы видеоадаптеров (Video Drivers) — реализуют графические операции;
  - драйверы потоковых устройств (Streaming Drivers) — осуществляют ввод/вывод потокового видео и звука;
  - WDM-драйверы (Windows Driver Model, WDM) — поддерживают технологию Plag and Play и управления электропитанием.

Замечу, что в эту классификацию я намеренно не включила драйверы новой драйверной модели WDF, т. к. считаю, что это будет уместнее сделать, когда уже оформятся окончательные версии как модели WDF, так и сопутствующих продуктов.

Конечно, рассмотреть все эти типы драйверов в одной книге мы не сможем. Главное — это дать направление и теоретическую и практическую подготовку, достаточные для дальнейшего самостоятельного освоения темы.

Далее стоит отметить, что драйверы бывают одно- и многоуровневыми. Если драйвер является многоуровневым, то обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами можно "поставить" любое количество фильтр-драйверов (filter-drivers). Также сейчас необходимо запомнить два термина — вышестоящие (higher-level) и нижестоящие (lower-level) драйверы. При обработке запроса данные идут от вышестоящих драйверов к нижестоящим, а при возврате результатов — наоборот. Ну и, понятно, одноуровневый (monolithic) драйвер просто является противоположностью многоуровневому.

Для технологии Plug and Play существуют три уровня-типа драйверов:

- ❒ шинные драйверы;
- ❒ фильтр-драйверы;
- ❒ функциональные драйверы.

На низшей ступени находится шинный драйвер, выше него — функциональный драйвер. Между и над ними находится определенное количество фильтр-драйверов. Если точнее, то:

1. Над шинным драйвером — фильтр-драйвер шины; эти два драйвера, очевидно, шинные.
2. Нижестоящие фильтр-драйвер устройства и классовый фильтр-драйвер.
3. Затем — собственно функциональный драйвер.
4. И, наконец, вышестоящие фильтр-драйвер устройства и классовый фильтр-драйвер; все драйверы со 2 по настоящий пункт относятся к драйверам устройства.

Напоминаю, что любой неясный вам термин вы можете посмотреть в словаре терминов, находящемся в *приложении 1.*

Упомянем о таком базисном понятии, как уровни запросов прерываний (IRQL).

Как известно, прерывания обрабатываются в соответствии с их приоритетом. В Windows NT используется особая схема прерываний, называемая

уровнями запросов прерываний. Всего уровней IRQL 32, самый низкий — 0 (passive), самый высокий — 31 (high). Прерывания с уровня 0 по 2 (DPC\dispatch) являются программными, а с 3 по 31 — аппаратными. Существуют специальные функции ядра, позволяющие узнать текущий уровень IRQL, а также сменить (понизить или повысить) его. Это довольно непростое, однако, дело, в котором есть множество своих нюансов (с каких уровней какие операции можно производить и т. д.) Но об этом подробнее не сейчас.

После того как мы более или менее разобрались с общими понятиями, мы уже можем приступить к обсуждению каких-то более сложных технологий. В частности, о технологии Plug and Play, которую я упоминала несколькими абзацами выше.

Технология Plug and Play (в условном переводе — "подключи и работай") — это технология, состоящая как из программной, так и из аппаратной поддержки механизма, позволяющего подключать/отключать, настраивать и т. д. применительно к системе все устройства, подключаемые к ней (конечно же, при условии, что подключаемые устройства поддерживают Plug and Play-технологию). В идеале весь этот процесс осуществляет только механизм Plug and Play, и какие-то действия со стороны пользователя вообще не требуются. Для каких-то устройств это так и происходит, для других — проблем, к сожалению, может быть гораздо больше. Кроме того, для успешной работы Plug and Play необходима не только поддержка этой технологии со стороны устройств, но также, конечно, со стороны драйверов и системного ПО.

Какие возможности предоставляет системное ПО (вместе с драйверами), поддерживающее технологию Plug and Play?

- ❒ автоматическое распознание подключенных к системе устройств;
- ❒ распределение и перераспределение ресурсов (таких как, например, порты ввода/вывода и участки памяти) между запросившими их устройствами;
- ❒ загрузка необходимых драйверов;
- ❒ предоставление драйверам необходимого интерфейса для взаимодействия с технологией Plug and Play;
- ❒ реализация механизма, позволяющего драйверам и приложениям получать информацию касаемо изменений в наборе устройств, подключенных к системе устройств, и совершить необходимые действия.

Главное перечислили. А теперь перейдем к рассмотрению структуры механизма Plug and Play.

Система Plug and Play состоит из двух компонентов, находящихся соответственно в пользовательском режиме и режиме ядра — менеджера Plug and Play пользовательского режима и менеджера Plug and Play "ядерного" режима.

Менеджер Plug and Play режима ядра работает с ОС и драйверами для конфигурирования, управления и обслуживания устройств. Менеджер Plug and Play пользовательского режима же взаимодействует с установочными компонентами пользовательского режима для конфигурирования и установки устройств. Также, при необходимости, менеджер Plug and Play взаимодействует с приложениями.

PnP (сокращенное обозначение Plug and Play) может успешно работать со следующими типами устройств:

- физические устройства;
- виртуальные устройства;
- логические устройства.

Об управлении питанием мы поговорим в *главе 3*, посвященной драйверной архитектуре WDM.

Какие условия драйвер должен выполнить для осуществления полной поддержки Plug and Play?

- наличие функции `DriverEntry`;
- наличие функции `AddDevice`;
- наличие функции `DispatchPnp`;
- наличие функции `DispatchPower`;
- наличие функции `Unload`;
- наличие cat-файла (файла каталога), содержащего сигнатуру WHQL;
- наличие inf-файла для установки драйвера.

Подробнее о технологии Plug and Play, об обработке PnP-запросов, о функциях, перечисленных в этом списке, об inf-файлах и т. д. я расскажу в *главах 3*, *5* и *6*, об архитектуре WDM, о структуре драйвера и, собственно, написании драйверов.

А сейчас сделаем небольшой обзор наиболее распространенных и полезных инструментов, используемых при написании драйверов.

1.2. Инструментарий

Описать и/или упомянуть обо всех утилитах, могущих понадобиться при разработке драйверов, — невозможно. Расскажу только об общих направлениях.

Без чего нельзя обойтись ни в коем случае — это Microsoft DDK (Driver Development Kit). К этому грандиозному пакету прилагается и обширная документация. Ее ценность — вопрос спорный. Но в любом случае хотя бы ознакомиться с первоисточником информации по написанию драйверов для Windows — обязательно. В принципе, можно компилировать драйверы и в Visual Studio, но для этого необходимо трудно и долго исправлять sln- и vcproj-файлы проектов для того, чтобы код вашего драйвера нормально компилировался. В любом случае исходные коды придется писать в Visual Studio, т. к. в DDK не входит полноценная интегрированная среда разработки (Integrated Development Environment, IDE). Есть пакеты разработки драйверов и от третьих фирм: WinDriver или NuMega Driver Studio, например. Но у них есть отличия базиса функций Microsoft (порой довольно большие) и масса других мелких неудобств. Так что DDK — лучший вариант. Для написания драйверов с использованием новейших технологий и нововведений Microsoft — априори KMDF и UMDF. Если же вы хотите писать драйверы исключительно на ассемблере, вам подойдет KmdKit (KernelMode Driver DevelopmentKit) для MASM32. Правда, этот пакет только для Windows 2000/XP.

Теперь можно поговорить о сторонних утилитах. Некоторые уже включены в стандартную поставку Windows: например, редактор реестра. Но их в любом случае не хватит, и многие программы нужно будет инсталлировать отдельно. Огромное количество таких программ создали патриархи системного программирования под Windows: Марк Руссинович, Гарри Нэббет, Свен Шрайбер и т. д. Марк Руссинович создал много полезных утилит: RegMon (рис. 1.1), FileMon (рис. 1.2) (мониторы обращений к реестру и файлам соответственно), WinObj (рис. 1.3) (средство просмотра каталогов имен объектов), DebugView (рис. 1.4), DebugPrint (программы просмотра, сохранения и т. д. отладочных сообщений) и проч., и проч. Все эти утилиты и огромное количество других можно найти на знаменитом сайте Руссиновича **http://www.sysinternals.com/**.

На диске, прилагающемся к известной книге "Недокументированные возможности Windows 2000" Свена Шрайбера [4], есть замечательные утилиты w2k_svc, -_sym, -_mem, позволяющие просматривать установленные драйверы, приложения и службы, работающие в режиме ядра, делать дамп памяти и т. д. Все эти утилиты, а также другие программы с диска, прилагающегося к книге, можно скачать с **http://www.orgon.com/w2k_internals/cd.html**.

Напоследок нельзя не упомянуть такие хорошие программы, как PE Explorer (рис. 1.5), PE Browse Professional Interactive (рис. 1.6), OllyDbg (рис. 1.7, 1.8), и такие незаменимые, как дизассемблер IDA (рис. 1.9, 1.10) и лучший отладчик SoftICE (рис. 1.11).

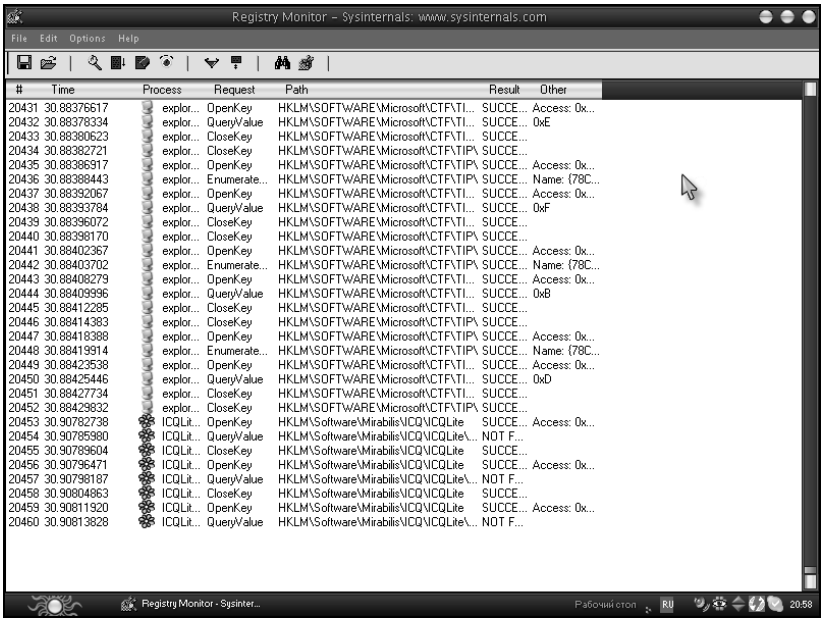

Рис. 1.1. Интерфейс программы RegMon

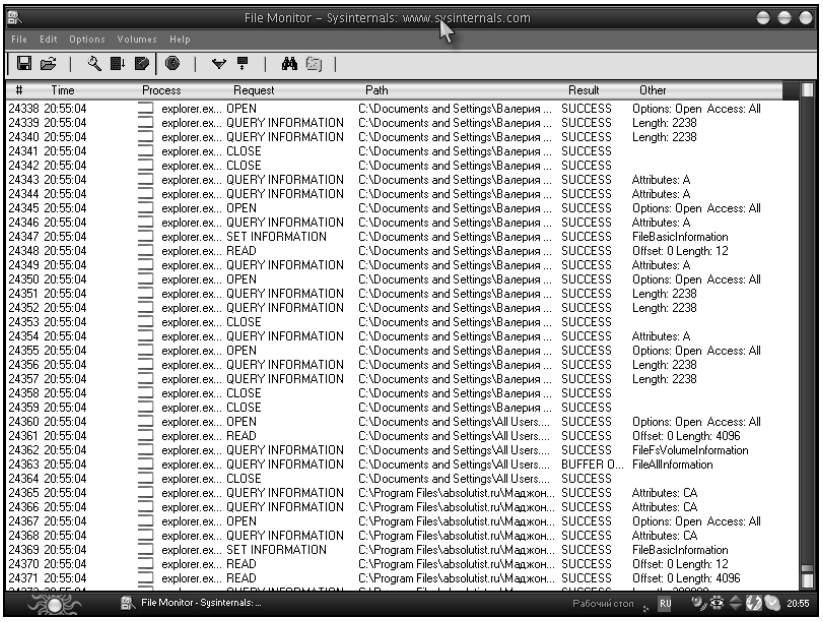

Рис. 1.2. Интерфейс программы FileMon

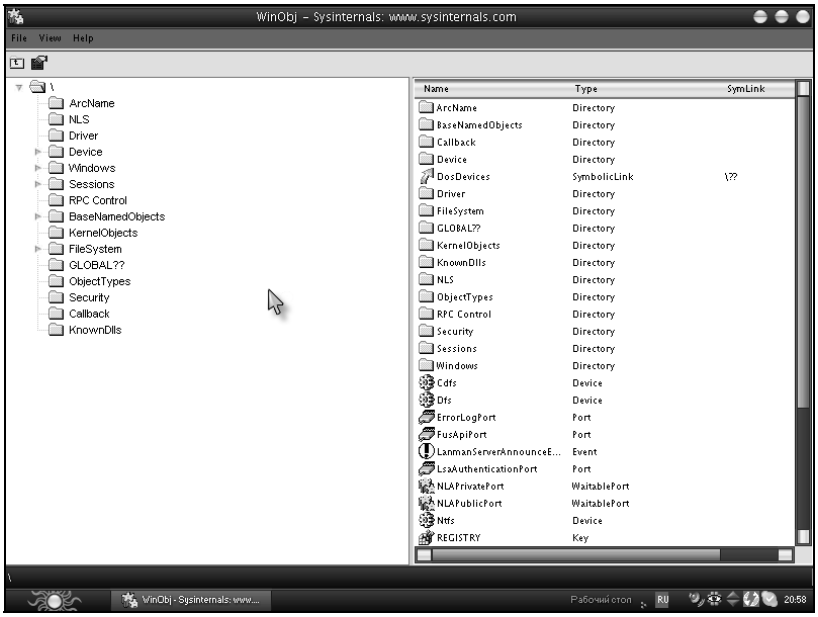

Рис. 1.3. Интерфейс программы WinObj

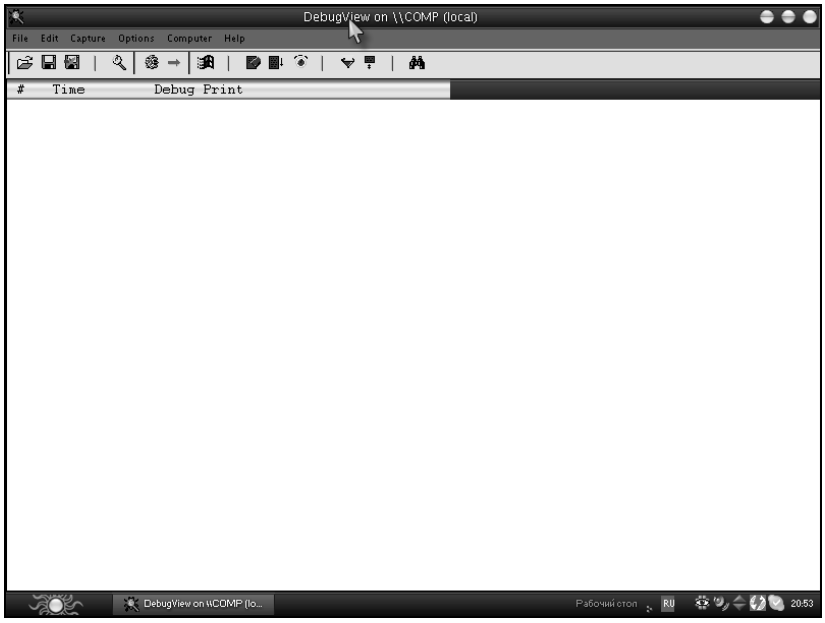

Рис. 1.4. Интерфейс программы DebugView

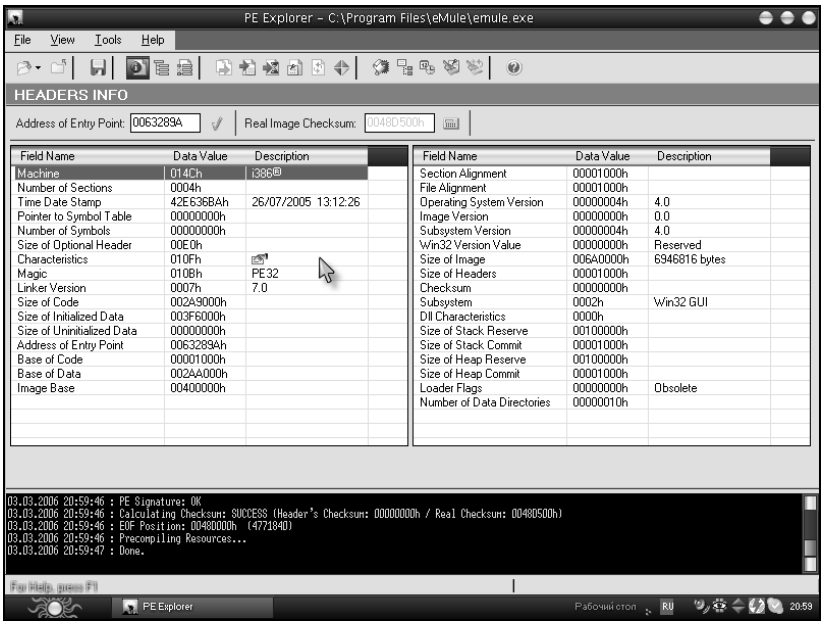

Рис. 1.5. Интерфейс программы PE Explorer

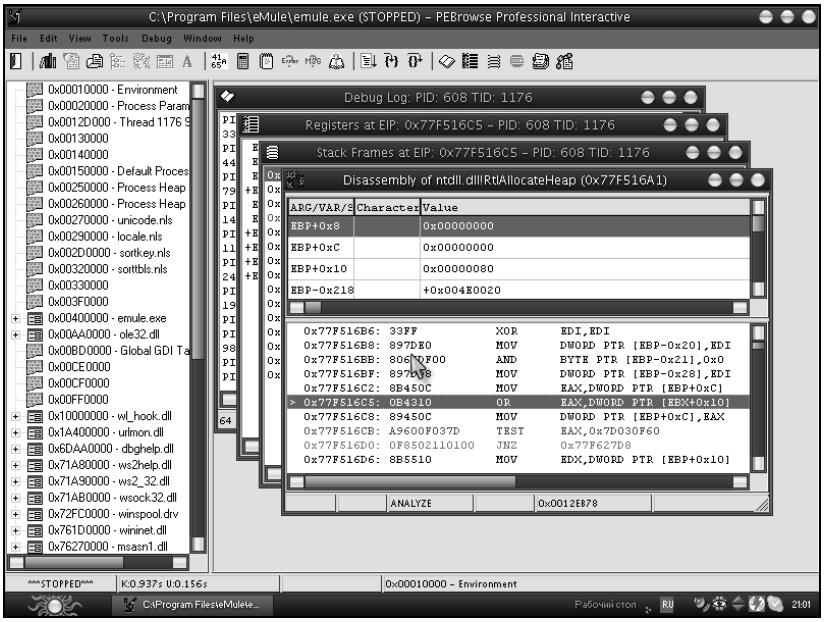

Рис. 1.6. Интерфейс PE Browse Professional Interactive

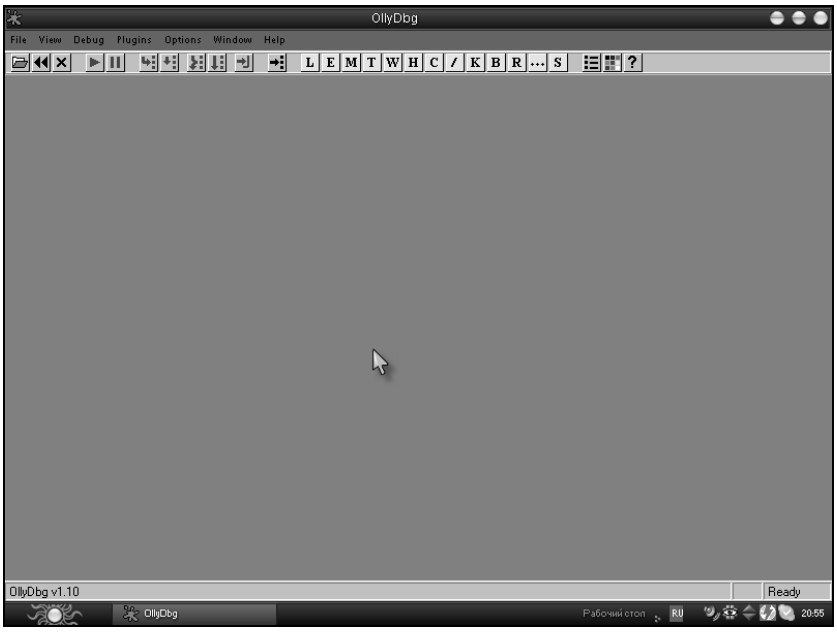

Рис. 1.7. Интерфейс программы OllyDbg при запуске

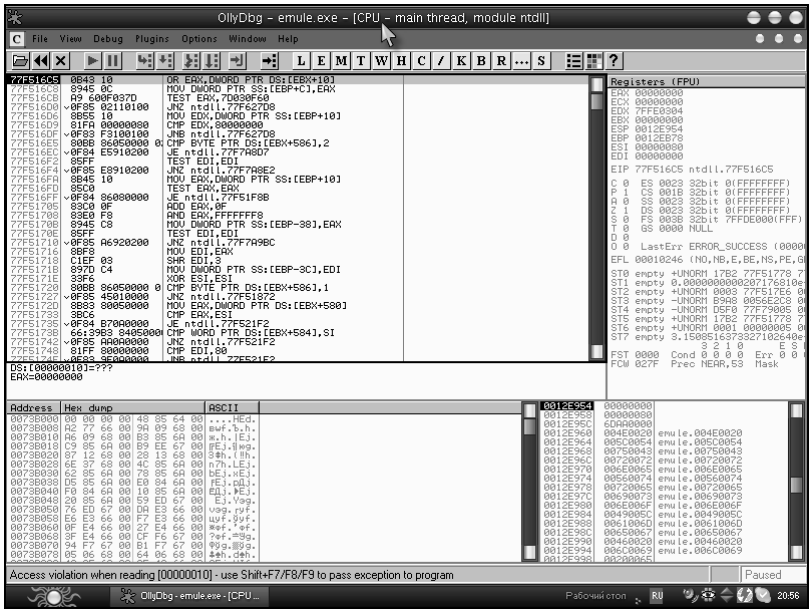

Рис. 1.8. Интерфейс программы OllyDbg с загруженным файлом

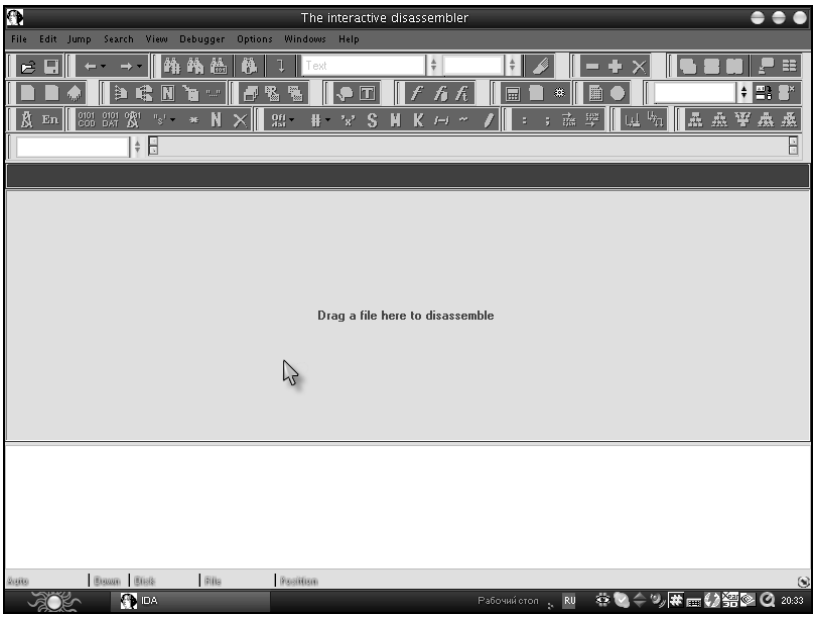

Рис. 1.9. Интерфейс программы IDA Professional при запуске

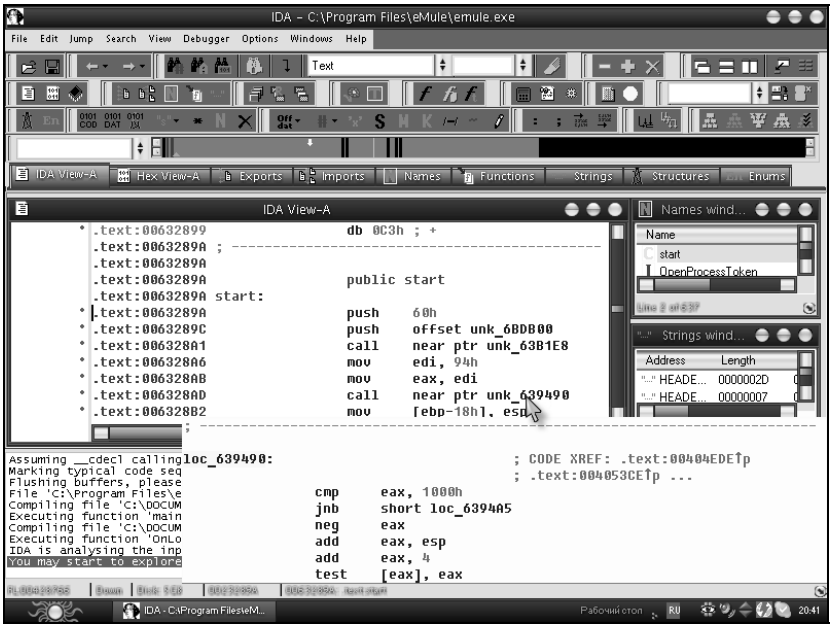

Рис. 1.10. Интерфейс программы IDA Professional с загруженным файлом

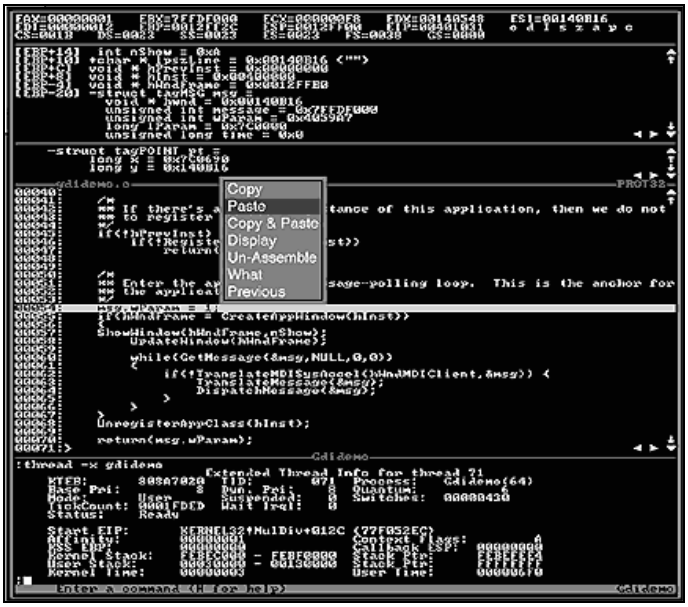

Рис. 1.11. Интерфейс программы SoftICE

* * *

Пожалуй, вводной информации пока достаточно. Перейдем к рассмотрению архитектуры Windows.

Глава 2

Архитектура Windows

В этой главе мы изучим наиболее важные и интересные особенности архитектуры ОС Windows.

Вот главные особенности операционной системы Windows семейства NT:

- ❒ модель измененного микроядра;
- ❒ возможность эмуляции нескольких ОС (наличие различных подсистем);
- ❒ многопоточность;
- ❒ интегрированная поддержка сети.

Наиболее важные из этих пунктов мы рассмотрим в этой главе (или же с отсылкой на другие).

Поговорим об уровнях разграничения привилегий, как об одном из важнейших моментов в архитектуре Windows NT. Я уже упоминала об User mode и Kernel mode. Эти два понятия тесно связаны с так называемыми кольцами. Их (колец) в Windows всего четыре: Ring 3, 2, 1 и 0. Ring 3 — наименее привилегированное кольцо, в котором есть множество ограничений по работе с устройствами, памятью и т. д. Например, в третьем кольце приложения не могут видеть адресное пространство других приложений без особого на то разрешения, выполнять привилегированные команды процессора, напрямую обращаться к оборудованию и т. д. В третьем кольце находится User mode. Kernel mode находится в нулевом кольце — наивысшем уровне привилегий. В этом кольце можно делать все: без всяких ограничений работать с системными данными и кодом, напрямую или через HAL обращаться к оборудованию... Вообще, в Kernel mode можно делать все, чего нельзя в User mode, и еще чуть-чуть. Процессоры Intel *x*86 поддерживают четыре уровня привилегий (четыре кольца), но Windows использует только два — 0 и 3. Понятно, что эти так называемые кольца определяются, прежде

всего, процессором (его аппаратными средствами). Поговорим чуть-чуть подробнее о режиме ядра.

Режим ядра (защищенный режим — по-другому) — это основной режим работы процессора (32-разрядного). Вот главные механизмы, реализуемые режимом ядра:

- механизм защиты памяти и ввода/вывода, состоящий из 4 уровней;
- механизм переключения задач;
- особая организация памяти. При этой организации памяти используются два различных способа ее преобразования: разбивка на страницы и сегментация;
- механизм защиты из четырех уровней — это уже упоминавшиеся выше 4 кольца.

Что такое переключение задач? Любая задача имеет состояние — иными словами (и с низкоуровневой точки зрения) состояние всех регистров процессора, с ней связанных (попросту — совокупность их значений). И состояние каждой задачи может быть сохранено. Где? Для этого есть специальные сегменты — сегменты состояния задач. Вот теперь и перейдем к разговору о двух механизмах преобразования памяти: сегментации и разбивке на страницы (страничная память, paging). Сначала о сегментации.

Что такое *сегмент*? Это отдельный блок общего пространства памяти. Максимальный размер сегмента — 4 Гбайт. Максимальное количество сегментов — 8192. Естественно, все эти цифры верны, только принимая во внимание использование 32-разрядной адресации.

Сегмент описывается особой структурой — *дескриптором*, размером в 8 байтов. В дескрипторе сегмента, в том числе, содержится информация о назначенных сегменту правах доступа (чтение, запись, чтение/запись) и назначенном уровне привилегий.

Сегментация обеспечивает неплохую защиту данных. Этому способствуют следующие ее особенности:

- исключается нарушение прав доступа;
- исключается обращение к сегменту без наличия нужного уровня привилегий;
- исключается обращение к элементам, находящимся за пределами сегмента (ошибочная адресация).

Ну, все, общее представление о сегментации получили. Перейдем к рассмотрению *страничного способа организации памяти*.

Главное то, что страничная организация памяти помогает использовать большее количество памяти, чем сегментация. Базируется она также на 32-

разрядной адресации, но в качестве базового объекта использует отдельный блок памяти размером 4 Кбайт.

Теперь вернемся к нашему самому первому списку и поговорим сначала о микроядре.

Итак, в чем заключается концепция микроядра? Есть программная база (очень маленькая), реализующая основные системные функции (примитивы). Это и есть *микроядро*, которое находится, конечно же, в привилегированном режиме. Все остальные компоненты ОС выполняются уже как отдельные системные процессы (не входящие в микроядро).

В чем плюсы и минусы использования такой технологии? Очевидные плюсы — легкость изменения и обновления всех компонентов ОС, выполненных в виде отдельных от микроядра системных процессов (т. к. эти изменения не затрагивают микроядро; также и наоборот). Главное, чтобы измененные компоненты при необходимости (если нет необходимости модифицировать и ядро тоже) экспортировали прежний интерфейс. Кроме того, это — в очень большой степени — залог устойчивости системы; если какие-либо компоненты ОС (отделенные от ядра) "упадут", то микроядро сможет сделать все возможное, чтобы без каких-либо сбоев в работе ОС перезагрузить эти компоненты.

При всех больших достоинствах использования архитектуры микроядра, конечно же, есть в этом и недостатки. Главный — низкая производительность архитектуры, использующей микроядро. Но для Windows NT в большой степени такой проблемы не существует, т. к. данная ОС использует измененный вариант этой архитектуры — архитектуру, использующую модифицированное микроядро.

Чем таким особенным отличается эта архитектура от обычной архитектуры микроядра? Тем, что теперь из "ядерного" режима в пользовательский перенесен целый набор подсистем, находящиеся в котором подсистемы (прошу прощения за повторение) делятся на два класса — подсистемы окружения и неотделимые (неделимые) подсистемы. Подробнее о подсистемах мы поговорим немного ниже.

Как работает при такой архитектуре прикладная программа? Прикладная программа работает с интерфейсом программирования, предоставляемым ей нужной подсистемой. Но, при необходимости, прикладная программа может использовать и свой интерфейс программирования.

Итак, как обстоят дела с пользовательским режимом при использовании этой архитектуры, мы более или менее разобрались. Что же находится в режиме ядра? В режиме ядра работает NT Executive (исполняющая система NT). Из чего она состоит? Из комплекта подсистем, микроядра и HAL. На-

бор подсистем и микроядро находятся в файле ntoskrnl.exe. HAL же находится (как можно интуитивно догадаться) в файле hal.dll.

А теперь рассмотрим архитектуру Windows NT (рис. 2.1).

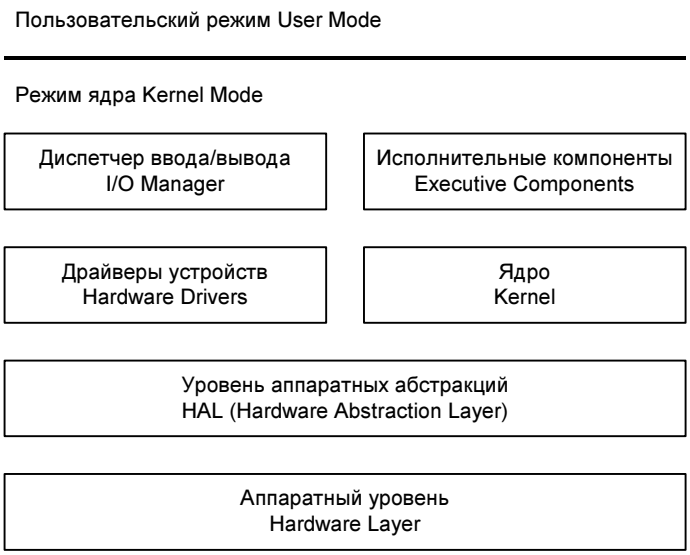

Рис. 2.1. Схема архитектуры Windows

Разберем эту схему поподробнее. С пользовательским режимом все понятно. В Kernel mode самый низкий уровень — аппаратный. Дальше идет HAL, выше — диспетчер ввода/вывода и драйверы устройств в одной связке, а также ядро вместе с исполнительными компонентами. Поподробнее поговорим об исполнительных компонентах (executive components). Что они дают? Прежде всего, они приносят пользу ядру. Как вы уже наверняка уяснили себе по схеме, ядро отделено от исполнительных компонентов. Возникает вопрос: почему? Просто на ядре оставили только одну задачу: простое управление потоками, а все остальные задачи (управление доступом, памятью для процессов и т. д.) берут на себя исполнительные компоненты. Они реализованы по модульной схеме, но несколько компонентов ее (схему) не поддерживают. Такая концепция имеет свои преимущества: таким образом, облегчается расширяемость системы.

Перечислю наиболее важные исполнительные компоненты:

- System Service Interface (Интерфейс системных служб);
- Configuration Manager (Менеджер конфигурирования);

- I/O Manager (Диспетчер ввода/вывода, ДВВ);
- Virtual Machine Manager, VMM (Менеджер виртуальных машин);
- Local Procedure Call, LPC (Локальный процедурный вызов);
- Process Manager (Диспетчер процессов);
- Object Manager (Менеджер объектов)

Вкратце расскажу о предназначении некоторых (наиболее важных/интересных) из них. System Service Interface дает приложениям пользовательского уровня возможность безопасно вызывать процедуры режима ядра. Local Procedure Call реализует механизм локальных вызовов между процессами на одном компьютере. Configuration Manager создает и хранит в единой базе данных (системном реестре) модель всего доступного аппаратного обеспечения и установленного программного обеспечения. Назначение диспетчеров процессов и ввода/вывода, а также менеджера объектов, я думаю, в пояснении не нуждается. Менеджер виртуальной памяти управляет выделением памяти в куче для кода режима ядра, выделением памяти для пользовательских приложений, виртуализацией запросов (создание иллюзии наличия свободной памяти путем выделения страниц на жестком диске (paging))... Одним словом, управляет памятью (от имени операционной системы).

Отложим пока в сторону наш главный список и отметим такое важное понятие, как в архитектуре ОС, так и в программировании вообще — об API.

API (Application Programming Interface) — это интерфейс прикладного программирования. Он позволяет обращаться прикладным программам к системным сервисам через их специальные абстракции.

API-интерфейсов несколько; таким образом, в Windows-системах присутствуют несколько подсистем.

- Подсистема Win32. Она отвечает за графический интерфейс пользователя, за обеспечение работоспособности Win32 API и за консольный ввод/вывод. Каждой реализуемой задаче соответствуют и свои функции: функции, отвечающие за графический интерфейс, за консольный ввод/вывод (GDI-функции), функции управления потоками, файлами и т. д.
- Подсистема VDM (Virtual DOS Machine, виртуальная DOS-машина). Задача подсистемы VDM (виртуальной DOS-машины) — эмулировать внутри ОС Windows NT для соответствующих приложений операционную систему MS-DOS.
- Подсистема POSIX (обеспечивает совместимость UNIX-программ). Подсистема POSIX делает то же самое, но только для POSIX-совместимых программ (только для них она, естественно, эмулирует не MS-DOS, а среду POSIX).

- Подсистема WOW (Windows on Windows). WOW 16 обеспечивает совместимость 32-разрядной системы с 16-битными приложениями. В 64-разрядных системах есть подсистема WOW 32, которая обеспечивает аналогичную поддержку 32-битных приложений.
- Подсистема OS/2. Обеспечивает совместимость с OS/2-приложениями.

Теперь, как я и обещала, поговорим подробнее о подсистемах (убежать от них уже просто некуда). Что вообще такое подсистема? *Подсистема* — это сервис, реализующий тот или иной комплект API, соответствующий той или иной операционной системе (поэтому есть подсистемы UNIX, DOS и т. д.). Главная подсистема — это, конечно, реализующая API-интерфейс самой ОС Windows — Win32. Подсистемы в NT основаны на клиент-серверной архитектуре.

Все остальные подсистемы (отличные от Win32), несмотря на то, что предоставляют свои собственные системы API, для работы с пользователем, конечно (такой, например, как отображение ему результатов), в любом случае используют подсистему Win32.

Как соотносятся ПО с подсистемами? Любая программа (так же, как и любой модуль) может работать только с одной из подсистем (как вариант — вообще ни с одной из них).

Естественно, в силу своей важности подсистема Win32 заслуживает того, чтобы поговорить о ней подробней. Так и поступим.

Подсистема Win32 состоит из двух кирпичиков — подсистемы среды и драйверов режима ядра. Подсистема среды отвечает за консольные окна, создание процессов, потоков и проч. Драйвер режима ядра поддерживает тоже множество вещей: и менеджер окон, и GDI, и т. д. Естественно, что все эти компоненты теснейшим образом связаны между собой.

И еще одна важная вещь — это NTDLL.DLL. Этот файл содержит особую систему, поддерживающую DLL-библиотеки. Поддерживает два типа функций: одна группа реализует интерфейс доступа к NT-службам, вторая группа — функции поддержки (APC, диспетчер исключений и т. д.).

* * *

Ну, все. Думаю, этого об архитектуре Windows достаточно. Перейдем к обсуждению архитектуры драйверной модели Windows — WDM.

Глава 3

Архитектура WDM

В предыдущих главах мы уже получили начальную теоретическую подготовку: разобрали основные термины и понятия, используемые в области программирования драйверов, поговорили об архитектуре Windows и о многих других вещах. Теперь настал момент, когда мы вплотную подошли к написанию драйверов. И начнем мы изучение этого процесса с WDM.

Что такое WDM? WDM (Windows Driver Model) — это драйверная модель от Microsoft для ОС Windows, пришедшая на смену предыдущей среде написания драйверов для ОС Windows — VxD (virtual device driver).

WDM в настоящий момент — одна из важнейших концепций в написании драйверов, разбираться в которой совершенно необходимо любому маломальски профессиональному разработчику драйверов под Windows. Поэтому не поговорить о ней и не разобраться в ней я считаю кощунством. Разберем архитектуру WDM, а в ее контексте изучим основные функции драйвера и их назначение.

Итак, мы уже разобрались, что WDM (Windows Driver Model) — это новая модель драйверов Windows. Ее главные особенности:

- совместимость на уровне двоичных кодов между драйверами для систем Windows 98 и NT;
- поддержка управления питанием (power management);
- поддержка Plug and Play;
- поддержка "продвинутого" шинного управления (advanced bus management).

Первый и последний пункты в пояснениях не нуждаются. Поговорим подробнее о втором.

Второй пункт — управление питанием. Технология управления питанием дает дополнительные возможности системе и драйверам устройств. Эти воз-

можности позволяют системе очень значительно сохранять электричество путем выборочного отключения питания нескольким или всем устройствам в системе. Технология управления питанием использует WMI (Windows Management and Instrumentation), и поэтому последний необходим в WDM-драйверах устройств. WMI — это набор специальных расширений к WDM-модели, предоставляющий интерфейс ОС, с помощью которого компоненты имеют возможность предоставлять различную информацию и оповещения.

Поговорим о компонентах режима ядра, обеспечивающих поддержку управления питанием:

- BIOS, поддерживающая ACPI;
- наличие драйвера ACPI;
- менеджер управления питанием;
- поддержка этого механизма со стороны драйвера (если драйверу это нужно, конечно).

С первыми двумя пунктами, думаю, все понятно. С третьим разберемся.

IRP-запрос, принадлежащий менеджеру питания, как легко догадаться, — `IRP_MJ_POWER`. С помощью этого запроса менеджер питания может установить новый режим питания или сменить уже существующий. Интерфейс, предоставляемый менеджером питания драйверу, — это функции вида `PoXxx`.

Продолжим наш разбор архитектуры WDM. Посмотрим в упрощенном виде на жизненный цикл среднестатистического WDM-драйвера:

1. Драйвер шины обнаруживает устройство.
2. PnP-Manager (Plug and Play-менеджер) определяет местонахождение ключа устройства в ветке `Enum` реестра. Этот ключ содержит указатель на другой ключ реестра, определяющий функциональный драйвер (который управляет отдельным устройством и является основным драйвером устройства) устройства. PnP-менеджер динамически загружает функциональный драйвер.
3. PnP-менеджер вызывает функцию драйвера `AddDevice` для того, чтобы создать `DRIVER_OBJECT`. Если драйвер соответствует больше, чем одному фактическому устройству, PnP-менеджер вызывает `AddDevice` для каждого из них. С этого момента вся коммуникация драйвера с внешним миром осуществляется с использованием IRP(I/O Request Packet)-пакетов.
4. PnP-менеджер выделяет все необходимые драйверу ресурсы ввода/вывода (запросы на прерывание, номера портов и т. д.) и посылает запрос для инициализации устройства.

5. Некоторые устройства могут быть удалены из системы без выключения компьютера. Если устройство — одно из таких, то PnP-менеджер посылает драйверу специальный IRP-пакет, в результате чего созданный функцией `AddDevice` объект устройства уничтожается.
6. Когда все устройства удалены, то менеджер ввода/вывода (I/O Manager) вызывает функцию `DriverUnload`, которая удаляет образ драйвера из памяти.

Очень важное место в архитектуре WDM занимает такое понятие, как Driver Layering, но здесь нам его понимание не жизненно важно, поэтому я подробно рассказываю о нем в словаре терминов в *приложении 1.*

Теперь рассмотрим главные функции драйвера в контексте архитектуры WDM.

WDM-функция `DriverEntry` (листинг 3.1) заполняет указатели на функции внутри объекта драйвера. Если на данном этапе нужны еще какие-либо глобальные инициализации, то функция `DriverEntry` выполняет их.

Листинг 3.1. Функция `DriverEntry`

```
NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT  DriverObject,
  IN PUNICODE_STRING  RegistryPath
);
{
  DriverObject->DriverUnload = DriverUnload;
  DriverObject->DriverExtension->AddDevice
    = AddDevice;
  DriverObject->MajorFunction[IRP_MJ_PNP]
    = DispatchPnp;
  DriverObject->MajorFunction[IRP_MJ_POWER]
    = DispatchPower;
  . . .
  return STATUS_SUCCESS;
}
```

Каждый драйвер должен обрабатывать PnP и запросы Power I/O. Для этого будут предназначены наши функции `DispatchPnp` и `DispatchPower`. Ну и, конечно, мы можем определить другие, необходимые нам функции.

Функция DriverUnload (листинг 3.2) — функция выгрузки драйвера должна очистить все глобальные инициализации, сделанные функцией DriverEntry (освободить ресурсы и т. д.).

Листинг 3.2. Функция DriverUnload

```
VOID
XxxUnload(
  IN PDRIVER_OBJECT  DriverObject
);
```

Отмечу, что в драйверах не-PnP функция DriverUnload выполняет заметно большую работу, чем таковая же в PnP-драйверах.

Далее приведу функцию AddDevice (листинг 3.3). Она пришла вместе с архитектурой WDM. Эту функцию система вызывает, чтобы уведомить о появлении устройства, для которого необходимо управление. PnP-менеджер вызывает эту функцию для каждого устройства, управляемого драйвером.

Листинг 3.3. Функция AddDevice

```
NTSTATUS
XxxAddDevice(
  IN PDRIVER_OBJECT  DriverObject,
  IN PDEVICE_OBJECT  PhysicalDeviceObject
);
```

Параметр DriverObject указывает на тот же самый объект драйвера, который мы проинициализировали в функции DriverEntry. Аргумент PDO — PhysicalDeviceObject — указывает на физический объект устройства, расположенный в самом низу стека устройств. Основные задачи AddDevice в функциональном драйвере — создать объект устройства и "прикрепить" его к стеку. Это реализуется следующей последовательностью шагов:

1. Вызываем функцию IoCreateDevice для создания объекта устройства и экземпляра объекта расширения устройства.
2. Регистрируем один или несколько интерфейсов устройства, для того чтобы приложения знали о его существовании. Возможен другой вариант: дайте объекту устройства имя и создайте на него символическую ссылку.
3. Инициализируем объект расширения устройства и поля Flags (флаги) объекта устройства.

4. Вызываем `IoAttachDeviceToDeviceStack` для включения нового устройства в стек устройств.

Поговорим о построении стека устройств. Каждый фильтр-драйвер и функциональный драйвер обязаны создавать стек объектов устройств, начиная с PDO и продолжая вверх. Все это легко реализуется с помощью функции `IoAttachDeviceToDeviceStack` (листинг 3.4).

Листинг 3.4. Функция `IoAttachDeviceToDeviceStack`

```
PDEVICE_OBJECT
IoAttachDeviceToDeviceStack(
  IN PDEVICE_OBJECT  SourceDevice,
  IN PDEVICE_OBJECT  TargetDevice
);
```

Как может быть использована эта функция внутри `AddDevice`, показано в листинге 3.5.

Листинг 3.5. Использование функции `IoAttachDeviceToDeviceStack` внутри `AddDevice`

```
NTSTATUS AddDevice ...
{
  PDEVICE_OBJECT fdo;
  IoCreateDevice(..., &fdo);
  pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
}
```

Первый аргумент функции `IoAttachDeviceToDeviceStack` — это адрес недавно созданного объекта устройства. Второй аргумент — это адрес PDO, который мы получили в качестве аргумента функции `AddDevice`. Возвращаемое значение — это адрес любого объекта устройства, который находится внизу сразу же после нашего. Это может быть PDO или же адрес какого-либо более низкого фильтр-объекта устройства.

Для обработки Plug and Play и Power IRP PnP-менеджер использует IRP для прямого старта, остановки и удаления устройств и для запрашивания драйверов об их устройствах. Все PnP IRP имеют главный функциональный код — `IRP_MJ_PNP`.

Драйверы должны обрабатывать PnP IRP в функции *Xxx*DispatchPnp, где *Xxx* — это префикс, идентифицирующий устройство. Драйвер устанавливает адрес функции DispatchPnp в DriverObject->MajorFunction[IRP_MJ_PNP] во время инициализации драйвера в его функции DriverEntry. PnP-менеджер, посредством I/O-менеджера, вызывает функцию DispatchPnp. Замечу, что все запросы управления питанием имеют IRP-код IRP_MJ_POWER.

Перейдем к инсталляции драйвера. Инсталлировать драйвер можно достаточно большим количеством способов. Здесь мы не будем их обсуждать, а вернемся к этой теме в главе, посвященной написанию простейшего драйвера *(см. разд. 6.3)*.

Все вышесказанное в этой главе относилось к обычным драйверам. А сейчас мы поговорим об особенном драйвере — фильтр-драйвере.

WDM фильтр-драйвер — это особый тип драйвера, который находится *над* каким-либо драйвером и перехватывает обращенные к нему запросы. Фильтр-драйвер работает посредством присоединения своего объекта устройства к объекту устройства более низкого драйвера. Фильтр-драйвер, который находится выше функционального драйвера, называется Upper Filter Driver, а тот, что находится ниже — Lower Filter Driver. Строение обоих типов драйверов практически одинаково. Применений фильтр-драйверам может быть много — перехват запросов ввода/вывода, дополнительная их обработка и т. д.

Рассмотрим скелет WDM фильтр-драйвера. Фильтр-драйвер, так же, как любой другой драйвер, имеет функции DriverEntry и AddDevice.

Функция DriverEntry фильтр-драйвера (листинг 3.6) практически идентична таковому в обычном драйвере. Но в отличие от обычного драйвера, фильтр-драйвер должен иметь функции для всех типов IRP, а не только для тех, которые собирается обрабатывать.

Листинг 3.6. Функция DriverEntry фильтр-драйвера

```
NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
{
  DriverObject->DriverUnload = DriverUnload;
  DriverObject->DriverExtension->AddDevice = AddDevice;
  for (int i = 0; i MajorFunction) ++i)
    DriverObject->MajorFunction[i] = DispatchAny;
  DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
```

```
  DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
  return STATUS_SUCCESS;
}
```

Функция `AddDevice` представлена в листинге 3.7. В этой функции мы вызовем `IoCreateDevice` для создания безымянного объекта устройства, а затем `IoAttachDeviceToDeviceStack` для включения его в стек. Также мы должны будем скопировать некоторые настройки объекта устройства, находящегося под объектом нашего драйвера.

Листинг 3.7. Функция `AddDevice` фильтр-драйвера

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
  PDEVICE_OBJECT fido;
  NTSTATUS status = IoCreateDevice(DriverObject,
    sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,
    0, FALSE, &fido);             // создаем устройство
  if (!NT_SUCCESS(status)) return status;
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  // расширение устройства
  {
    pdx->DeviceObject = fido;
    pdx->Pdo = pdo;
    IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 255);
    PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceStack(fido, pdo);
    // присоединяем устройство к стеку
    pdx->LowerDeviceObject = fdo; // указатель на более низкий объект
    fido->Flags |= fdo->Flags & (DO_DIRECT_IO | DO_BUFFERED_IO |
                                 DO_POWER_PAGABLE | DO_POWER_INRUSH);
    fido->DeviceType = fdo->DeviceType;
    fido->Characteristics = fdo->Characteristics;
    fido->AlignmentRequirement = fdo->AlignmentRequirement;
    fido->Flags &= ~DO_DEVICE_INITIALIZING;
  }
}
```

Припомним, что фильтр-драйвер используется в основном для того, чтобы, так сказать, изменить поведение устройства. Для этого нам нужны функции, которые что-либо делают с приходящими IRP, прежде чем передать их дальше. Мы будем просто передавать большинство IRP вниз по стеку. Рассмотрим главную функцию обработки IRP-функции — `DispatchAny` (листинг 3.8).

Функция `DispatchAny` предназначена для обработки всех IRP, за исключением `IRP_MJ_PNP` и `IRP_MJ_POWER`.

Листинг 3.8. Функция `DispatchAny` фильтр-драйвера

```
NTSTATUS DispatchAny(PDEVICE_OBJECT fido, PIRP Irp)
{
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
  IoSkipCurrentIrpStackLocation(Irp); // копируем параметры IRP-стека
                                      // с текущей позиции
                                      // до вышестоящего драйвера
  status = IoCallDriver(pdx->LowerDeviceObject, Irp); // посылаем запрос
                                                      // вышестоящему драйверу
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
}
```

`IoAcquireRemoveLock` препятствует удалению объекта устройства в то время, когда мы обращаемся к полям внутри него и внутри объекта расширения устройства, "прикрепленного" к нему. `CompleteRequest` — это вспомогательная функция, которая работает с механикой завершения IRP.

Теперь поговорим об обработке отдельных IRP: `IRP_MJ_PNP` и `IRP_MJ_POWER`.

Сначала — обработка `IRP_MJ_PNP` (листинг 3.9).

Листинг 3.9. Функция `DispatchPnp` фильтр-драйвера для обработки `IRP_MJ_PNP`

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fido, PIRP Irp)
{
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
```

```
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
  // получаем текущую позицию стека
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG fcn = stack->MinorFunction;
  IoSkipCurrentIrpStackLocation(Irp);
  status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  if (fcn == IRP_MN_REMOVE_DEVICE)
  {
    IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
    IoDetachDevice(pdx->LowerDeviceObject); // отсоединяем устройство
    IoDeleteDevice(fido);                   // удаляем устройство
  }
  else IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
}
```

Теперь — обработка `IRP_MJ_POWER` (листинг 3.10).

Листинг 3.10. Функция `DispatchPower` фильтр-драйвера для обработки `IRP_MJ_POWER`

```
NTSTATUS DispatchPower(PDEVICE_OBJECT fido, PIRP Irp)
{
  PoStartNextPowerIrp(Irp); // информируем Power Manager о том,
                            // что драйвер готов обрабатывать
                            // следующий запрос IRP-Power
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
  IoSkipCurrentIrpStackLocation(Irp);
  status = PoCallDriver(pdx->LowerDeviceObject, Irp); // отправляет
                            // запрос к вышестоящему драйверу
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
}
```

Два главных отличия от функции `DispatchAny` в том, что мы должны вызвать функции `PoStartNextPowerIrp` в момент получения IRP (это нужно сделать в любом случае) и `PoCallDriver` (вместо `IoCallDriver`), чтобы передать IRP следующему драйверу.

* * *

Все. Пока этого общего обзора архитектуры WDM нам будет достаточно. В этой главе мы затронули главные функции WDM-драйвера. Продолжим обсуждение этой темы в следующей главе, целиком посвященной структуре драйвера.

Глава 4

Программирование в режиме ядра

Основной тип драйверов, который рассматривается в данной книге — это драйверы режима ядра. Поэтому нам необходимо получить ясное представление о программировании в этом режиме. Почему это необходимо? Дело в том, что программирование в режиме ядра имеет массу особенностей, для прикладных программистов очень непривычных, а для новичков — довольно сложных. Во-первых, у режима ядра свое, отличное от такового в пользовательском режиме, API. Кроме того, для кода, выполняющегося в режиме ядра, имеет очень большое значение его уровень IRQL, т. к. приложениям, выполняющимся на высоких уровнях IRQL, недоступны многие функции, к которым имеют доступ приложения низких IRQL-уровней, и наоборот. Все это необходимо учитывать. Во-вторых, в режиме ядра есть свои дополнительные описатели типов. Полный их список можно найти в заголовочном файле ntdef.h. Его содержание примерно такое, как представлено в листинге 4.1.

Листинг 4.1. Заголовочный файл ntdef.h

```
typedef unsigned char USHAR
typedef unsigned short USHORT
typedef unsigned long ULONG
.............
```

Зачем это нужно? Во-первых, для красоты; ну, а если серьезно, то для унификации стиля классических C-типов данных и нововведенных — таких как `WCHAR` (двухбайтный Unicode-символ), `LARGE_INTEGER` (который на самом

деле является объединением) и т. д. А также для унификации исходных кодов для 32-разрядных платформ и уже пришедших 64-разрядных.

В исходных кодах драйверов часто встречаются макроопределения IN, OUT, OPTIONAL. Что они означают? А ровным счетом ничего, и введены они только для повышения удобства чтения исходника. OPTIONAL обозначает необязательные параметры, IN — параметры, передаваемые внутрь функции, например, OUT — соответственно, наоборот. А вот IN OUT означает, что параметр передается внутрь функции, а затем возвращается из нее обратно.

Есть изменения и в типах возвращаемых значений функций. Вы наверняка знаете, что в C функции либо не возвращают значения (void), либо возвращают значение определенного типа (char, int и т. д.). При программировании драйверов вы столкнетесь с еще одним типом — NT_STATUS. Этот тип включает в себя информацию о коде завершения функции (определение этого типа можно посмотреть в файле ntdef.h). NT_STATUS является переопределенным типом long integer. Неотрицательные значения переменных этого типа соответствуют успешному завершению функции, отрицательные — наоборот (файл NTSTATUS содержит символьные обозначения всех кодов возврата). Сообщение об удачном завершении имеет код 0 и символьное обозначение STATUS_SUCCESS. Остальные коды возврата, соответствующие разнообразным вариантам ошибок, транслируются в системные коды ошибок и передаются вызывающей программе. Для работы с типом NT_STATUS существует несколько макроопредений (описанные в файле ntdef.h), например, NT_SUCCESS(), проверяющий код возврата на успешность завершения.

Функции драйвера, за исключением DriverEntry, могут называться, как угодно. Тем не менее, существуют определенные "правила хорошего тона" при разработке драйверов, в том числе и для именования процедур: например, все функции, относящиеся к HAL, желательно предварять префиксом HAL и т. д. Разработчики корпорации Microsoft практически постоянно следуют этому правилу. А имена типов данных и макроопределения в листингах DDK написаны сплошь заглавными буквами. Советую вам поступать также при разработке своих драйверов. Это и в самом деле во много раз повышает удобство работы с листингом и, к тому же, показывает уровень вашего профессионализма всем, кто работает с вашим листингом.

А теперь, чтобы вам стали более или менее понятны основные различия между программированием в пользовательском и системном режимах, расскажу о некоторых функциях режима ядра для работы с памятью, реестром и строками UNICODE_STRING.

Начнем с функций для работы с памятью, но сперва поговорим собственно об устройстве и работе с памятью в Windows. Единое 4-гигабайтное адресное пространство памяти Windows (я имею в виду 32-разрядные версии

Windows) делится на две части: 2 Гбайт для пользовательского пространства и 2 Гбайт для системного. 2 Гбайт системного пространства доступны для всех потоков режима ядра. Системное адресное пространство делится на следующие части (рис. 4.1).

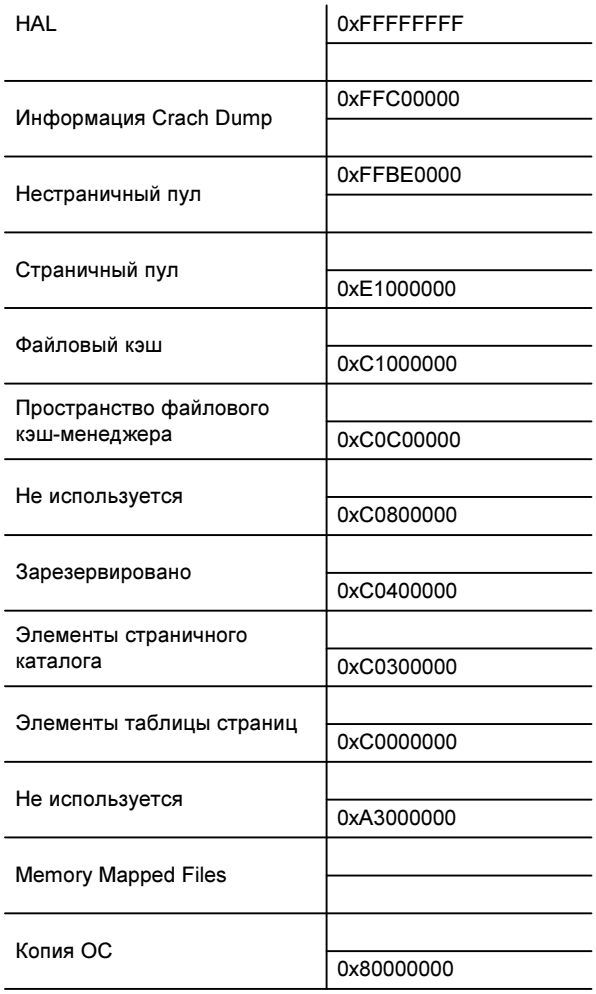

Рис. 4.1. Схема адресного пространства Windows

Видов адресов в режиме ядра три: *физические* (реально указывающие на область физической памяти), *виртуальные* (которые перед использованием транслируются в физические) и *логические* (используемые HAL-уровнем при

общении с устройствами; он же и отвечает за работу с такими адресами). Функции режима ядра, отвечающие за выделение и освобождение виртуальной памяти, отличаются от таковых в пользовательском режиме. Также, находясь на уровне режима ядра, становится возможным использовать функции выделения и освобождения физически непрерывной памяти. Разберем все эти функции поподробнее.

- `PVOID ExAllocatePool` (уровень IRQL, на котором может выполняться функция, меньше `DISPATCH_LEVEL`) — выделяет область памяти. Принимает два параметра: параметр `POOL_TYPE`, в котором содержится значение, указывающее, какого типа область памяти нужно выделить: `PagedPool` — страничная, `NonPagedPool` — нестраничная (в этом случае функцию можно вызвать с любого IRQL-уровня). Второй параметр (`ULONG`) — размер запрашиваемой области памяти. Функция возвращает указатель на выделенную область памяти, и `NULL`, если выделить память не удалось.
- `VOID ExFreePool` (`IRQL<DISPATCH_LEVEL`) — освобождает область памяти. Принимает параметр `PVOID` — указатель на освобождаемую область памяти. Если высвобождается нестраничная память, то данная функция может быть вызвана с `DISPATCH_LEVEL`. Возвращаемое значение — `void`.
- `PVOID MmAllocateContiguousMemory` (`IRQL==PASSIVE_LEVEL`) — выделяет физически непрерывную область памяти. Принимает два параметра. Первый параметр (`ULONG`) — размер запрашиваемой области памяти, второй параметр — `PHYSICAL_ADDRESS`, означающий верхний предел адресов для запрашиваемой области. Возвращаемое значение — виртуальный адрес выделенной области памяти или `NULL` (при неудаче).
- `VOID MmFreeContiguousMemory` (`IRQL==PASSIVE_LEVEL`) — освобождает физически непрерывную область памяти. Принимает единственный параметр (`PVOID`) — указатель на область памяти, выделенную ранее с использованием функции `MmAllocateContiguousMemory`. Возвращаемое значение — `void`.
- `BOOLEAN MmIsAddressValid` (`IRQL<=DISPATCH_LEVEL`) — делает проверку виртуального адреса. Принимает параметр `PVOID` — виртуальный адрес, нуждающийся в проверке. Функция возвращает `TRUE`, если адрес допустимый (т. е. присутствует в виртуальной памяти), и `FALSE` — в противном случае.
- `PHYSICAL_ADDRESS MmGetPhysicalAddress` (`IRQL` — любой) — определяет физический адрес по виртуальному. Принимает параметр `PVOID`, содержащий анализируемый виртуальный адрес. Возвращаемое значение — полученный физический адрес.

Основные функции для работы с памятью рассмотрели, перейдем к таковым для работы с реестром. Сначала поговорим о функциях доступа к реестру,

предоставляемых диспетчером ввода/вывода, потом о драйверных функциях прямого доступа к реестру, а затем о самом богатом по возможностям и удобству семействе функций для работы с реестром — Zw~.

Итак, перечислю драйверные функции, предоставляемые диспетчером ввода/вывода.

- `IORegisterDeviceInterface` — данная функция регистрирует интерфейс устройства. Диспетчер ввода/вывода создает подразделы реестра для всех зарегистрированных интерфейсов. После этого можно создавать и хранить в этом подразделе нужные драйверу параметры с помощью вызова функции `IoOpenDeviceInterfaceRegistryKey`, которая возвращает дескриптор доступа к подразделу реестра для зарегистрированного интерфейса устройства.
- `IoGetDeviceProperty` — данная функция запрашивает из реестра установочную информацию об устройстве.
- `IoOpenDeviceRegistryKey` — возвращает дескриптор доступа к подразделу реестра для драйвера или устройства по указателю на его объект.
- `IoSetDeviceInterfaceState` — с помощью данной функции можно разрешить или запретить доступ к зарегистрированному интерфейсу устройства. Компоненты системы могут получать доступ только к разрешенным интерфейсам.

А теперь драйверные функции для прямого доступа к реестру.

- `RtlCheckRegistryKey` — проверяет, существует ли указанный подраздел внутри подраздела, переданного первым параметром. Что и каким образом передавать в первом параметре — в рамках книги все не перечислить, отсылаю к ntddk.h и wdm.h. Если существует — возвращается `STATUS_SUCCESS`.
- `RtlCreateRegistryKey` — создает подраздел внутри раздела реестра, указанного вторым параметром. Далее — все то же самое, что и у `RtlCheckRegistryKey`.
- `RtlWriteRegistryValue` — записывает значение параметра реестра. Первый параметр — куда пишем, второй — в какой подраздел (если его нет, то он будет создан), а третий — какой параметр создаем.
- `RtlDeleteRegistryValue` — удаляет параметр из подраздела. Параметры те же самые, что и у `RtlWriteRegistryValue` (только с необходимыми поправками, конечно).
- `RtlQueryRegistryValues` — данная функция позволяет за один вызов получить значения сразу нескольких параметров указанного подраздела.

И напоследок функции для работы с реестром семейства Zw~.

- `ZwCreateKey` — открывает доступ к подразделу реестра. Если такового нет — создает новый. Возвращает дескриптор открытого объекта.
- `ZwOpenKey` — открывает доступ к существующему подразделу реестра.
- `ZwQueryKey` — возвращает информацию о подразделе.
- `ZwEnumerateKey` — возвращает информацию о вложенных подразделах уже открытого ранее подраздела.
- `ZwEnumerateValueKey` — возвращает информацию о параметрах и их значениях открытого ранее подраздела.
- `ZwQueryValueKey` — возвращает информацию о значении параметра в открытом ранее разделе реестра. Полнота возвращаемой информации определяется третьим параметром, передаваемым функции, который может принимать следующие значения (дополнительные разъяснения не требуются, т. к. они имеют "говорящие" имена): `KeyValueBasicInformation`, `KeyValuePartialInformation` и `KeyValueFullInformation`.
- `ZwSetValueKey` — создает или изменяет значение параметра в открытом ранее подразделе реестра.
- `ZwFlushKey` — принудительно сохраняет на диск изменения, сделанные в открытых функциями `ZwCreateKey` и `ZwSetValueKey` подразделах.
- `ZwDeleteKey` — удаляет открытый подраздел из реестра.
- `ZwClose` — закрывает дескриптор открытого ранее подраздела реестра, предварительно сохранив сделанные изменения на диске.

Практически все вышеперечисленные функции для работы с реестром должны вызываться с уровня IRQL `PASSIVE_LEVEL`.

Теперь о функциях режима ядра для работы со строками `UNICODE_STRING`.

- `NTSTATUS RtlAppendUnicodeStringToString` (может вызываться с любого уровня IRQL) — эта функция объединяет строки `UNICODE_STRING`. Первый параметр `(PUNICODE_STRING)` — это указатель на строку-получатель, а второй `(PUNICODE_STRING)` — это указатель на присоединяемую строку. Возвращает `STATUS_SUCCESS` в случае успеха и `STATUS_BUFFER_TOO_SMALL`, если размер двухбайтового буфера строки-получателя слишком мал.
- `LONG RtlCompareUnicodeString (IRQL==PASSIVE_LEVEL)` — выполняет сравнение двух строк `UNICODE_STRING`. Принимает параметры `PUNICODE_STRING` — указатели на первую и вторую сравниваемые строки и параметр `BOOLEAN`, равный `TRUE`, если нужно игнорировать регистр. Возвращает 0, если сравниваемые строки равны, и меньше 0 — если первая строка меньше второй.

- `BOOLEAN RtlEqualUnicodeString (IRQL==PASSIVE_LEVEL)` — во всем аналогична предыдущей функции, за исключением типа возвращаемого значения. Возвращает `TRUE`, если строки идентичны, и `FALSE` — если они различны.
- `NTSTATUS RtlInt64ToUnicodeString (IRQL==PASSIVE_LEVEL)` — данная функция выполняет преобразование `int64` в `UNICODE_STRING`. Исходное число принимает в параметре типа `ULONGLONG`, в параметре `ULONG` — обозначение формата (0 (10) — десятичный, 2 — двоичный, 8 — восьмеричный, и 16 — шестнадцатеричный), и в параметре `PUNICODE_STRING` — строку `UNICODE_STRING`.

  Возвращает `STATUS_SUCCESS` в случае успеха, `STATUS_BUFFER_OVERFLOW`, если буфер `UNICODE_STRING` слишком маленький, и `STATUS_INVALID_PARAMETER`, если параметр `Base` содержит ошибочное значение.
- `NTSTATUS RtlUnicodeStringToAnsiString (IRQL==PASSIVE_LEVEL)` — преобразует `UNICODE_STRING` в `ANSI_STRING`. Принимает два очевидных параметра — `PANSI_STRING` и `PUNICODE_STRING`, которые содержат, соответственно, указатель на выходную строку `ANSI_STRING` и исходную `UNICODE_STRING`-строку. Еще один параметр типа `BOOLEAN` равен `TRUE`, если требуется выделить память под буфер строки `ANSI_STRING`. В случае успеха возвращает `STATUS_SUCCESS`, а в случае неудачи — код ошибки.

* * *

Думаю, пока достаточно. Конечно, у всех вышеперечисленных функций есть масса нюансов в применении. Да и вообще функций режима ядра — великое множество, их ничуть не меньше, чем в пользовательском режиме. Но моя задача была не рассказать обо всех API-функциях режима ядра (что даже в рамках книги сделать довольно затруднительно, тем более, что это не центральная тема книги), а продемонстрировать отличия функций режима ядра от таковых в пользовательском режиме и хоть немного рассказать о нюансах их применения. Взять, к примеру, то, что в пользовательском режиме не имеет значения, в потоке какого приоритета будет выполняться приложение: оно будет иметь такой же полный доступ ко всем API-функциям пользовательского режима, как и любые другие приложения; на уровне ядра, как вы только что убедились, это не так. Ну, а за более или менее полным списком и описанием всех этих API-функций советую обратиться к библии Гарри Нэббета [2]. Ну, вот и все, теперь вы готовы к разговору о структуре драйвера, который мы сейчас и начнем.

Глава 5

Структура драйвера

В этой главе мы рассмотрим структуру простейшего драйвера режима ядра. Хочу отметить, что в данной главе есть определенное количество информации, в той или иной степени перекрывающееся уже сказанным в этой книге или же тем, что будет сказано позже. Я сочла это необходимым, для того чтобы дать целостное представление о структуре драйвера и избавить читателя от необходимости "рыскать" по разным "углам" книги. Кроме того, напоминаю в очередной раз, что в конце книги есть словарь всех терминов, упоминающихся в книге *(см. приложение 1)*. Если какой-то термин будет вам непонятен, вы всегда можете справиться о нем в словаре. Разъяснение терминов прямо здесь, в самом тексте, опять-таки мешает целостному восприятию текста и заставляет отвлекаться на множество несущественных для темы данной главы деталей (которые сами по себе могут быть, тем не менее, очень важными).

Фактически драйвер можно представить как довольно-таки обычную DLL-библиотеку уровня ядра. Таким образом, далее можно представить драйвер просто как набор процедур, периодически вызываемых внешними программами. Несмотря на то, что процедуры драйверов для разных устройств сильно отличаются, есть общая структура и общие функции для всех драйверов. Главные из них мы сейчас и рассмотрим. Все нижеперечисленные функции составляют так называемый "скелет", на основе которого строится любой драйвер — каким бы сложным он ни был.

Первая — и самая главная — это функция инициализации драйвера: `DriverEntry` (листинг 5.1).

Листинг 5.1. Функция `DriverEntry`

```
NTSTATUS
DriverEntry(
```

```
  IN PDRIVER_OBJECT  DriverObject,
  IN PUNICODE_STRING  RegistryPath
);
```

`DriverEntry` — ключевая функция драйвера. Ее главные задачи — произвести все необходимые действия по инициализации и определить точки входа для остальных функций драйвера. Эта функция вызывается при загрузке драйвера. Вышеприведенный код — прототип этой функции. Как видим, она принимает два аргумента — два указателя. Первый аргумент — указатель на объект `DriverObject` типа `PDRIVER_OBJECT`. Он позволяет функции `DriverEntry` определить указатели на функции `Dispatch`, `AddDevice`, `StartIo`, а также на функцию выгрузки драйвера в объекте драйвера. Аргумент `RegistryPath` передает функции `DriverEntry` указатель на Unicode-строку, содержащую путь к ключу драйвера в реестре (\Registry\Machine\System\CurrentControlSet\Services*DriverName*). Каждый драйвер должен иметь инициализационную процедуру, которую менеджер ввода/вывода вызывает автоматически, если эта процедура называется `DriverEntry`.

Каждый драйвер должен иметь, по крайней мере, одну процедуру `Dispatch` (листинг 5.2).

Листинг 5.2. Процедура `Dispatch`

```
NTSTATUS
XxxDispatchPnP(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP  Irp
);
```

Если драйвер устройства не может завершить все возможные запросы ввода/вывода в его `Dispatch`-процедуре, он должен иметь либо процедуру `StartIo` (листинг 5.3), либо заводить одну или более внутренних очередей и управлять собственным механизмом отложенных запросов на прерывание.

Листинг 5.3. Процедура `StartIo`

```
VOID
XxxStartIo(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP  Irp
);
```

В зависимости от уровня, занимаемого драйвером в стеке обработки запроса на прерывание, драйвер может обладать следующими процедурами.

Вдобавок к процедуре `DriverEntry` драйвер может иметь процедуру `Reinitialize` (листинг 5.4), вызываемую один или несколько раз в процессе загрузки системы после того, как `DriverEntry` вернет управление.

Листинг 5.4. Процедура `Reinitialize`

```
VOID
Reinitialize(
  IN PDRIVER_OBJECT  DriverObject,
  IN PVOID  Context,
  IN ULONG  Count
);
```

Любой драйвер физического устройства, который генерирует прерывания, должен иметь эту процедуру (листинг 5.5). Этот драйвер всегда самый низкий в стеке.

Листинг 5.5. Процедура `InterruptService`

```
BOOLEAN
InterruptService(
  IN PKINTERRUPT  Interrupt,
  IN PVOID  ServiceContext
);
```

Любой драйвер, имеющий ISR, должен иметь `DpcForIsr` (листинг 5.6) или `CustomDpc` (листинг 5.7).

Листинг 5.6. Процедура `DpcForIsr`

```
OID
DpcForIsr(
  IN PKDPC  Dpc,
  IN struct _DEVICE_OBJECT  *DeviceObject,
  IN struct _IRP  *Irp,
  IN PVOID  Context
);
```

Листинг 5.7. Процедура `CustomDpc`

```
VOID
CustomDpc(
  IN struct _KDPC  *Dpc,
  IN PVOID  DeferredContext,
  IN PVOID  SystemArgument1,
  IN PVOID  SystemArgument2
);
```

Любой низкоуровневый драйвер устройства, данные которого или регистры сопряженного устройства могут изменяться в его ISR и других процедурах драйвера, должен иметь одну или более процедур `SynchCritSection` (листинг 5.8).

Листинг 5.8. Процедура `SynchCritSection`

```
BOOLEAN
SynchCritSection(
  IN PVOID  SynchronizeContext
);
```

Любой драйвер устройства, использующий DMA, должен иметь процедуру `AdapterControl` (листинг 5.9). Любой драйвер устройства, который должен синхронизировать операции с физическим контроллером для нескольких устройств или каналов устройства, должен иметь `ControllerControl` (листинг 5.10).

Листинг 5.9. Процедура `AdapterControl`

```
IO_ALLOCATION_ACTION
AdapterControl(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP   Irp,
  IN PVOID  MapRegisterBase,
  IN PVOID  Context
);
```

Листинг 5.10. Процедура `ControllerControl`

```
IO_ALLOCATION_ACTION
ControllerControl(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP   Irp,
  IN PVOID  MapRegisterBase,
  IN PVOID  Context
);
```

Клавиатура, мышь, последовательный, параллельный, звуковой драйверы и драйвер файловой системы имеют процедуру `Cancel` (листинг 5.11). Любой драйвер, обрабатывающий запрос в течение длительного промежутка времени (когда пользователь может отменить операцию), должен иметь процедуру `Cancel`. Обычно эту процедуру имеет высший драйвер в стеке обработки запроса.

Листинг 5.11. Процедура `Cancel`

```
VOID
Cancel(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP  Irp
);
```

Любой драйвер верхнего уровня, который создает запросы к более низкоуровневым драйверам, должен иметь, по крайней мере, одну процедуру `IoCompletion` для освобождения всех структур IRP, созданных драйвером. Таким образом, любой драйвер высшего уровня должен иметь процедуру `IoCompletion` (листинг 5.12). Другие процедуры драйвера могут сказать, чтобы `IoCompletion` была вызвана, когда все низкоуровневые драйверы обработают текущий запрос.

Листинг 5.12. Процедура `IoCompletion`

```
NTSTATUS
IoCompletion(
  IN PDEVICE_OBJECT  DeviceObject,
```

```
  IN PIRP   Irp,
  IN PVOID  Context
);
```

Для отслеживания времени, занимаемого процедурой ввода/вывода, или для некоторой другой цели, определяемой разработчиком, любой драйвер должен иметь процедуры `IoTimer` (листинг 5.13) и/или `CustomTimerDpc` (листинг 5.14). `IoTimer` вызывается раз в секунду, когда драйвер включает таймер. Процедура `CustomTimerDpc` может быть вызвана в более мелкий или переменный интервал.

Листинг 5.13. Процедура `IoTimer`

```
VOID
IoTimer(
  IN struct DEVICE_OBJECT  *DeviceObject,
  IN PVOID  Context
);
```

Листинг 5.14. Процедура `CustomTimerDpc`

```
VOID
CustomTimerDpc(
  IN struct _KDPC  *Dpc,
  IN PVOID  DeferredContext,
  IN PVOID  SystemArgument1,
  IN PVOID  SystemArgument2
);
```

Драйвер должен иметь процедуру `Unload` (листинг 5.15), если он может быть выгружен во время работы системы.

Листинг 5.15. Процедура `Unload`

```
VOID
XxxUnload(
  IN PDRIVER_OBJECT  DriverObject
);
```

Для осуществления обращений к микроядру, работы с реестром, памятью, объектами, синхронизацией и пр. существует набор функций, называющихся *функциями поддержки ядра*. Я приведу только самые необходимые. Для дополнительной информации обращайтесь к Microsoft Windows NT DDK.

Функция `IoCreateDevice` создает новый объект устройства и инициализирует его для использования драйвером (листинг 5.16). Объект устройства представляет собой физическое, виртуальное или логическое устройство, которое необходимо драйверу для поддержки динамического управления этим устройством.

Листинг 5.16. Функция `IoCreateDevice`

```
NTSTATUS
IoCreateDevice(
  IN PDRIVER_OBJECT DriverObject, // указатель на объект драйвера
  IN ULONG DeviceExtensionSize,   // размер блока пользовательской
                                  // информации в байтах
  IN PUNICODE_STRING DeviceName,  // имя устройства (иногда опускается)
  IN DEVICE_TYPE DeviceType,      // тип устройства (последовательное,
                                  // диск, мышь и т. д.)
  IN ULONG DeviceCharacteristics, // параметры устройства
                                  // (вынимаемое и пр.)
  IN BOOLEAN Exclusive,           // параллельность доступа к устройству
  OUT PDEVICE_OBJECT *DeviceObject  // указатель на объект создаваемого
                                    // устройства;
```

Функция `IoCreateSymbolicLink` создает символическую ссылку между устройством и видимым пользователем именем (листинг 5.17).

Листинг 5.17. Функция `IoCreateSymbolicLink`

```
NTSTATUS
IoCreateSymbolicLink(
  IN PUNICODE_STRING SymbolicLinkName, // символическое имя,
                                       // видимое пользователем
  IN PUNICODE_STRING DeviceName     // имя устройства
                                    // в пространстве имен ядра Windows
);
```

Функция `IoCompleteRequest` объявляет менеджеру ввода/вывода, что обработка текущего запроса ввода/вывода закончена (листинг 5.18).

Листинг 5.18. Функция `IoCompleteRequest`

```
VOID
IoCompleteRequest(
  IN PIRP Irp            // указатель на запрос ввода/вывода
  IN CCHAR PriorityBoost // повышение приоритета драйвера для обработки
  // запроса. Зависит от обрабатываемого устройства.
  // IO_NO_INCREMENT при ошибке или очень быстрой обработке запроса
);
```

* * *

Здесь, конечно, перечислено далеко не все и не полностью. Но эта глава — все-таки не справочник, а просто обучающий текст, старающийся дать необходимую базу.

Теперь мы можем приступить к написанию нашего первого драйвера.

Глава 6

Простейший драйвер для Windows

В этой главе мы поэтапно рассмотрим процесс написания простейшего не-WDM-драйвера для ОС Windows, с оглядкой на дальнейшее развитие наших навыков и увеличение познаний в программировании драйверов. Будут рассмотрены собственно написание кода, компиляция, установка и отладка простейшего драйвера.

6.1. Написание драйвера

Вот и пришло время написания простейшего драйвера под Windows. Мы разобрали особенности архитектуры Windows NT, поговорили об особенностях драйвера, как понятия, и о его структуре, познакомились с некоторыми приемами программирования в режиме ядра. А теперь мы, наконец, вплотную подошли к, собственно, написанию своего первого (или тридцать первого) драйвера. В данном разделе мы это и осуществим. Мы напишем простейший legacy-драйвер (драйвер в стиле NT), скомпилируем и установим его. И в результате в диспетчере устройств Windows, наконец-то, появится "устройство", драйвер к которому будет написан вами. Приступим к воплощению этой мечты в реальность!

Процесс работы над нашим драйвером мы начнем, естественно, с написания кода (писать будем, как я уже говорила, на C). Назовем наш драйвер MyDriver, приступим и создадим файл MyDriver.c.

Записывая код этого драйвера, давайте будем сразу приучаться писать его красиво: грамотно разделяя код по файлам, называя функции в соответствии со всеми принятыми в сообществе программистов драйверов соглашениями.

Что будет делать наш драйвер? Он будет очень простым и "бесполезным" — просто будет выводить сообщения — и больше ничего.

Важное примечание: код простейшего драйвера — единственный, который не будет дан полностью; ни в самой книге, ни на компакт-диске. Сделано это для того, чтобы у читателя не было искушения схитрить и просто скопировать код драйвера и откомпилировать. Процесс самостоятельного "думанья" — совершенно необходим для грамотного и хорошего понимания принципов и процессов программирования драйверов. В книге дана абсолютно *вся* информация (и даже с избытком) для успешного дописывания этого драйвера самостоятельно. Тем более, что замысел драйвера — элементарен.

По ходу всего приведенного кода я буду давать необходимые комментарии.

Приступим. Итак, файл MyDriver.c (листинг 6.1).

Листинг 6.1. Файл MyDriver.c. Заголовочные определения

```
#include "ntddk.h" // к нашему стыду, мы пока пишем не-WDM-драйвер;
                   // поэтому и подключим главный заголовочный файл не
                   // WDM.h, а ndddk.h
#include "MyDriverMessages.h"  // здесь содержатся нужные нам константы
```

Итак, любой драйвер должен иметь определенное количество подключенных заголовочных файлов. Они могут быть обязательными и не очень. В данном случае обязательный заголовочный файл — это ntddk.h, который нужен любому legacy-драйверу; а остальные два — это заголовочные файлы, которые мы сами создали, потому что этого требует наш замысел.

А теперь определим все необходимые драйверу функции (листинг 6.2).

Листинг 6.2. Определения всех необходимых драйверу функций

```
NTSTATUS DriverEntry        // дадим определения всем основным функциям
                            // драйвера; DriverEntry – процедура
                            // инициализации и загрузки
(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath);
NTSTATUS CreateMyDriver   (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
// определим функцию создания драйвера
NTSTATUS ReadMyDriver     (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
// определим функцию его чтения
NTSTATUS WriteMyDriver    (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
```

```
// определим функцию его записи
NTSTATUS ShutdownMyDriver (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
// определим функцию отключения драйвера
NTSTATUS CleanupMyDriver  (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
// определим функцию очистки
NTSTATUS IoCtlMyDriver    (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
// определим функцию обработки IOCTL-запросов
VOID MyDriverUnload (IN PDRIVER_OBJECT DriverObject);
// определим функцию выгрузки драйвера
BOOLEAN GetMessage(IN NTSTATUS ErrorCode, IN PVOID IoObject,IN PIRP Irp);
// определение функции вывода сообщений
#ifdef ALLOC_PRAGMA // Эта директива проверяет наличие определения
                    // механизма изолирования странично организованного
                    // кода в отдельные именованные секции. Это просто
                    // пример для одной функции. Вообще, каждую функцию
                    // можно вынести в секцию. В нашем случае мы сделали
                    // это только для функции вывода сообщений.
#pragma alloc_text(PAGE, GetMessage)
#endif
```

В нашем случае у нас и определения, и реализации функций находятся в одном файле — ввиду простоты драйвера. На деле хорошим тоном считается "разнос" определений и реализаций функций по разным файлам.

Рассмотрим реализацию функции `DriverEntry` (листинг 6.3).

Листинг 6.3. Реализация функции `DriverEntry`

```
NTSTATUS DriverEntry   // пишем реализацию функции DriverEntry
(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
UNICODE_STRING nameString, linkString; // определяем две строки в формате
                                       // Unicode
PDEVICE_OBJECT deviceObject;           // определяем переменную объекта
                                       // устройства
NTSTATUS status;                       // определяем переменную кода
                                       // статуса
```

```
GetMessage(MSG_DRIVER_ENTRY,DriverObject,NULL); // вызываем функцию
                                                // вывода сообщения
RtlInitUnicodeString(&nameString, L"\\Device\\MyDriver");
// инициализация нашей строки Unicode
status = IoCreateDevice(DriverObject,sizeof(65533),&nameString,
          0, 0, FALSE, &deviceObject); // создаем устройство
if (!NT_SUCCESS(status))               // проверяем статус
  return status;
deviceObject->Flags |= DO_DIRECT_IO;    // устанавливаем флаги
deviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
RtlInitUnicodeString(&linkString, L"\\DosDevices\\MyDriver");
// инициализация другой нашей строки
status = IoCreateSymbolicLink (&linkString, &nameString); // создаем
                                                // символьную ссылку
if (!NT_SUCCESS(status))   // проверяем статус
{
  IoDeleteDevice (DriverObject->DeviceObject); // удаляем устройство
  return status;
}
DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateMyDriver;
   // проставляем указатели на функции
DriverObject->MajorFunction[IRP_MJ_READ] = ReadMyDriver;
DriverObject->MajorFunction[IRP_MJ_WRITE] = WriteMyDriver;
DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] = ShutdownMyDriver;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoCtlMyDriver;
DriverObject->DriverUnload=MyDriverUnload;
return STATUS_SUCCESS;
. . .
}
```

Итак, я привела одну из возможных реализаций функции `DriverEntry` в нашем драйвере.

Далее — функция создания драйвера (листинг 6.4).

Листинг 6.4. Функция `CreateMyDriver` создания драйвера

```
NTSTATUS CreateMyDriver (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
  GetMessage(MSG_CREATE,(PVOID)DeviceObject,NULL);// выводим сообщения
  IoCompleteRequest(Irp, IO_NO_INCREMENT);        // завершаем обработку
                                                  // запроса
  return STATUS_SUCCESS;
. . .
}
```

Теперь — функция чтения (листинг 6.5). Здесь можете выполнять, что захотите — не суть важно...

Листинг 6.5. Функция `ReadMyDriver` чтения

```
NTSTATUS ReadMyDriver(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
  . . .
  GetMessage(MSG_READ,DeviceObject,NULL);
  . . .
  return STATUS_SUCCESS;
}
```

Теперь — функция записи (листинг 6.6). Здесь тоже можете выполнять, что захотите — тоже не суть важно...

Листинг 6.6. Функция `WriteMyDriver` записи

```
NTSTATUS WriteMyDriver(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
  . . .
  GetMessage(MSG_WRITE,DeviceObject,NULL);
  . . .
  return STATUS_SUCCESS;
  . . .
}
```

Следующая — функция выключения (остановки) драйвера (листинг 6.7).

Листинг 6.7. Функция `ShutdownMyDriver` остановки

```
NTSTATUS ShutdownMyDriver(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
  . . .
  GetMessage(MSG_SHUTDOWN,DeviceObject,NULL);
  IoCompleteRequest(Irp, IO_NO_INCREMENT); // завершаем обработку запроса
  return STATUS_SUCCESS;
  . . .
}
```

Функция обработки IOCTL-запросов (листинг 6.8).

Листинг 6.8. Функция `IoCtlMyDriver` обработки IOCTL-запросов

```
NTSTATUS IoCtlMyDriver(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
  . . .
  GetMessage(MSG_IOCTL,DeviceObject,NULL);
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}
```

Функция выгрузки драйвера представлена в листинге 6.9.

Листинг 6.9. Функция `MyDriverUnload` выгрузки драйвера

```
VOID MyDriverUnload (IN PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING linkString; // создаем Unicode-строку ...
  GetMessage(MSG_DRIVERUNLOAD, DriverObject, NULL);

  // и инициализируем ее
  RtlInitUnicodeString (&linkString, L"\\DosDevices\\MyDriver");
```

```
  IoDeleteSymbolicLink (&linkString);   // удаляем символьную ссылку
  IoDeleteDevice(DriverObject->DeviceObject);  // удаляем устройство
}
```

И, наконец — гвоздь программы — функция вывода сообщений (листинг 6.10), которую мы так часто вызывали по всему коду драйвера (если быть точной — в каждой функции).

Листинг 6.10. Функция `GetMessage` вывода сообщений

```
BOOLEAN GetMessage(IN NTSTATUS ErrorCode, IN PVOID IoObject,IN PIRP Irp)
{
  PIO_ERROR_LOG_PACKET Log_Packet;     // пакет журналирования ошибок
  PIO_STACK_LOCATION IrpStackLocation; // местоположение стека
  PWCHAR pInsertString;
  STRING AnsiInsertString;
  UNICODE_STRING UniInsertString;
  UCHAR Size_of_Packet;                // размер пакета
  Size_of_Packet = sizeof(IO_ERROR_LOG_PACKET);
  Log_Packet = IoAllocateErrorLogEntry(IoObject, Size_of_Packet);
  if (Log_Packet == NULL) return FALSE;
  Log_Packet ->ErrorCode = ErrorCode;
  Log_Packet ->UniqueErrorValue = 0,
  Log_Packet ->RetryCount = 0;
  Log_Packet ->SequenceNumber = 0;
  Log_Packet ->IoControlCode = 0;
  Log_Packet ->DumpDataSize = 0;
  if (Irp!=NULL)
  {
    IrpStack=IoGetCurrentIrpStackLocation(Irp);
    Log_Packet ->MajorFunctionCode = IrpStack->MajorFunction;
    Log_Packet ->FinalStatus = Irp->IoStatus.Status;
  }
  else
  {
    Log_Packet ->MajorFunctionCode = 0;
```

```
    Log_Packet ->FinalStatus = 0;
  }
  IoWriteErrorLogEntry(Log_Packet);
  return TRUE;
}
```

Здесь назначения параметров интуитивно понятны, поэтому не будем на них останавливаться.

Напоследок создадим файл с сообщениями. Как это делается?

Прежде всего, создадим файл MyDriverMessages.mc с примерно таким содержанием, которое приведено в листинге 6.11.

Листинг 6.11. Файл MyDriverMessages.mc

```
MessageID    = 1
Severity     = Informational
SymbolicName = MSG_DRIVER_ENTRY
Language     = English
Driver Entry
.
MessageID    = 2
Severity     = Informational
SymbolicName = MSG_CREATE
Language     = English
Create
.

MessageID    = 3
Severity     = Informational
SymbolicName = MSG_READ
Language     = English
Read
.
...
```

И т. д. Механизм ясен. Далее этот файл необходимо сохранить с расширением mc и откомпилировать с помощью утилиты `mc` (от англ. *message compiler*):

```
mc MyDriverMessages.mc
```

Message compiler создаст файл MyDriverMessages.rc (который мы подключим чуть-чуть позднее) и файл MyDriverMessages.h (который мы уже подключили выше).

Между прочим, такой драйвер, как мы только что написали, можно использовать в качестве основы для каких-либо других драйверов отладочных версий, т. к. использование нашей функции `GetMessage` — достаточно красивый и удобный способ отображения отладочных сообщений.

С написанием кода разобрались. Перейдем к следующим этапам работы над драйвером.

6.2. Компиляция драйвера

Скомпилировать драйвер можно двумя способами: в Visual Studio и в DDK. Первый способ хорош тем, что в Visual Studio можно набрать (при этом, как обычно, будет производиться автоматическая проверка синтаксиса кода) и скомпилировать там же код. Но для того, чтобы происходила проверка и корректная компиляция кода, необходимо, как я уже говорила, исправить sln-файлы проекта, что достаточно трудно и вообще дурной тон. Способ же с использованием DDK проще и надежнее, поэтому на данный момент разберем второй вариант. Для компиляции и сборки драйвера в DDK (с использованием утилиты Build) необходимо создать два файла: Makefile и sources. Первый управляет работой Build и в нашем случае имеет следующий стандартный вид (листинг 6.12).

Листинг 6.12. Файл Makefile

```
# DO NOT EDIT THIS FILE!!! Edit .\sources. if you want to add a new source
# file to this component. This file merely indirects to the real make file
# that is shared by all the driver components of the Windows NT DDK
#
!INCLUDE $ (NTMAKEENV) \ makefile.def
#
```

Файл source содержит в себе индивидуальные настройки процесса компиляции и сборки драйвера. В нашем случае он будет выглядит так, как показано в листинге 6.13.

Листинг 6.13. Файл sources

```
TARGETNAME=MYDRIVER // имя компилируемого драйвера
TARGETTYPE=DRIVER   // тип компилируемого проекта
#DRIVERTYPE=WDM     // при компиляции WDM-драйвера эту строку нужно
// раскомментировать, а в заголовочном файле Driver.h
// вместо ntddk.h подключить wdm.h
TARGETPATH=obj // каталог, в котором будут размещены промежуточные файлы
SOURCES=MyDriver.c MyDriverMsg.rc // файлы исходных текстов
C_DEFINES=-DUNICODE -DSTRICT
```

Теперь все необходимые для компиляции файлы (в нашем случае — MyDriver.c, файлы, созданные message compiler, Makefile, sources) осталось только поместить в один каталог и запустить компиляцию отладочной (checked) версии драйвера с помощью утилиты Build. Все, компиляция и сборка драйвера завершены. Перейдем к инсталляции.

6.3. Инсталляция драйвера

Инсталлировать драйвер можно несколькими способами:

- с внесением записей в реестр;
- с использованием программы Monitor из пакета Driver Studio;
- с помощью inf-файла;
- с использованием SCM-менеджера (программно) (к слову сказать, не всегда есть такое богатство выбора — WDM-драйверы, например, рекомендуется инсталлировать только с помощью inf-файла и мастера установки оборудования).

Мы рассмотрим первые три способа, а о четвертом я вкратце скажу несколько слов.

Наш драйвер без проблем инсталлируется и работает как под Windows 9*x*, так и под Windows NT (секрет этого заключается в драйвере ntkern.vxd из Windows 9*x*, который помогает NT-драйверам "почувствовать себя, как дома"; но, естественно, возможности его не безграничны), но процесс записи в реестр (и записываемые значения) немного отличаются. Разберем оба варианта.

Запустите стандартный Блокнот Windows, наберите в нем строки из листинга 6.14 и сохраните документ под любым именем в виде reg-файла.

Листинг 6.14. Файл MyDriver.reg

```
# содержимое файла реестра, необходимого для инсталляции драйвера,
# под Windows 9x:
REGEDIT4
[HKEY_LOCAL_MACHINE\System\ CurrentControlSet\Services\MyDriver]
"ErrorControl"=dword:00000001
"Type" =dword:00000001
"Start" =dword:00000002
"ImagePath" ="\\SystemRoot\\System32\\Drivers\MyDriver.sys"
```

Названия параметров говорят сами за себя, так что, думаю, дополнительных пояснений не требуется. Для инсталляции драйвера в Windows NT необходимо практически то же самое. Откройте реестр по тому же пути, который был указан в вышеприведенном reg-файле (не важно, вручную, или же создавая reg-файл), создайте тот же раздел и те же параметры со значениями 1, 1 и 2 соответственно. Понятно, что перед внесением изменений в реестр готовый драйвер нужно поместить в каталог, указанный в параметре `ImagePath`.

Программа Monitor из пакета DriverStudio позволяет загрузить, запустить, остановить и удалить драйвер и имеет интуитивно понятный графический интерфейс, работе с которым, я думаю, обучать не нужно. Перед запуском драйвера из Monitor можно предварительно запустить программу DebugView — тогда все отладочные сообщения драйвера будут выдаваться в ее окно.

Рассмотрим способ инсталляции драйвера с помощью inf-файла.

Прежде всего: что такое inf-файл? Inf-файл — это текстовый файл, в котором содержится вся важная информация обо всех устройствах и/или файлах, устанавливаемых с помощью этого inf-файла. От обычного текстового файла он отличается (структурно) тем, что разделен на именованные секции. Какие-то из этих секций имеют строго определенные имена и не могут быть изменены, какие-то создатель inf-файла может называть так, как ему больше нравится. Секции могут располагаться в inf-файле абсолютно в любом порядке.

Каждая секция начинается со своего имени. Оно заключено в квадратные скобки `[]`. Если вы назовете несколько секций одним и тем же именем, то система всех их объединит в одну. Названия секций, директив и т. д. не чувствительны к регистру (таким образом, `section`, `Section` и `SECTION` — суть одни и те же имена).

Комментарии в inf-файлах определены символом ;.

Вообще, можно еще очень долго перечислять мелкие особенности синтаксического строения inf-файлов. Поступим по-другому. Возьмем из DDK пример inf-файла. Я же буду каждую важную строку этого inf-файла снабжать необходимыми комментариями (листинг 6.15).

Листинг 6.15. Файл inf

```
[Version]     ; Начало секции Version. Начинать inf-файл принято
              ; именно с этой секции. Кроме того, эта секция
              ; должна быть в каждом inf-файле.
Signature="$Windows NT$" ; Сигнатура. Может быть одной из трех —
                         ; $Windows NT$, $Chicago$ или $Windows 95$.
Class=Mouse   ; Определяет имя класса для любого стандартного типа
              ; устройств.
              ; Стандартный вариант — использование системных
              ; имен классов, перечисленных в файле devguid.h.
ClassGUID={4D36E96F-E325-11CE-BFC1-08002BE10318} ; Определяет GUID
  ; (Global Unique Identifier) для данного класса устройств.
Provider=%Provider%   ; Принято присваивать этому параметру переменную,
                      ; значение которой указано в секции String
                      ; (секции строк).
LayoutFile=layout.inf ; Этот параметр используется только системным
                      ; установщиком.
DriverVer=09/28/1999,5.00.2136.1 ; Дата драйвера и его версия.


[DestinationDirs]    ; В этой секции указывается папка (папки)
                     ; назначения для всех операций создания
                     ; (копирования и т. д.) над всеми файлами,
                     ; упоминающихся в этом inf-файле.
DefaultDestDir=12    ; Данный вариант указывает каталог по умолчанию —
                     ; каталог драйвера.

; ... [ControlFlags] обычно в этой секции находится несколько параметров
; (ExcludeFromSelect) для определения того, какие устройства,
```

```
; перечисленные в секции Models, которые не будут отображены
; пользователю, как доступные во время ручной установки.

[Manufacturer]     ; Эта секция определяет производителей устройств,
                   ; устанавливаемых при помощи этого inf-файла.
                   ; Также эта секция определяет имя секции Models.

%StdMfg% = StdMfg  ; (Standard types)
%MSMfg% = MSMfg    ; Microsoft

[StdMfg]  ; Секция Models производителя (standard)
          ; Std - мышь последовательного порта.
%*pnp0f0c.DeviceDesc% = Ser_Inst,*PNP0F0C,SERENUM\PNP0F0C,SERIAL_MOUSE
; Мышь Std InPort
%*pnp0f0d.DeviceDesc% = Inp_Inst,*PNP0F0D
     ; ... здесь может быть необходимое количество Std-полей

     ; Секции DDInstall (Ser_Inst, Inp_Inst и другие), определяемые
     ; моделью, также расположены здесь.
     ; Секции DDInstall описывают способы, варианты и т. д. установки

[Strings] ; Необходимые строки, используемые в inf-файле.
; Идентификатор %strkey% определяет строки, видимые пользователю.
Provider = "Microsoft"
; ...
StdMfg   = "(Standard mouse types)"
MSMfg    = "Microsoft"

; ...
*pnp0f0c.DeviceDesc   = "Standard Serial Mouse"
*pnp0f0d.DeviceDesc   = "InPort Adapter Mouse"
; ...
HID\Vid_045E&Pid_0009.DeviceDesc = "Microsoft USB Intellimouse"
; ...
```

Для того чтобы мы могли корректно установить наш драйвер с использованием мастера установки (путь к созданному inf-файлу нужно указать при инсталляции с помощью мастера установки Windows), нам необходимо создать inf-файл с правильной структурой. Начинаться он может, например, так, как представлено в листинге 6.16.

Листинг 6.16. Файл MyDriver.inf

```
; MyDriver.Inf
[Version]
Signature="$Windows NT$"
Class=Unknown
Provider=%Provider%
DriverVer=01/11/2006,0.0.0.1
. . .
[Strings]
Provider = "Home"
. . .
```

И т. д. Все как в эталонном inf-файле.

Сохраняйте этот файл с расширением inf. Ну, а теперь можете запускать мастер установки, в нем указывайте способ выбора устройства вручную (без автоматического поиска и определения), путь к каталогу с inf-файлом... Ну, думаю, вас не нужно учить общению с мастером. После завершения установки, если все пройдет нормально, то драйвер успешно установится в систему, и вы сможете видеть его в списке устройств. Все. Теперь его можно тестировать.

Перейдем к SCM-менеджеру. SCM-менеджер — это сервис Windows NT, предоставляющий удобную возможность работать с драйвером без использования мастера установки — с помощью функций, вызываемых из пользовательского приложения. Для того чтобы начать работать с SCM-менеджером, необходимо вызвать функцию `OpenSCManager`, а для завершения работы — функцию `CloseServiceHandle`. Но, к сожалению, работать с помощью SCM-менеджера можно не со всеми типами драйверов.

Еще один способ установки — использование соинсталлятора. Соинсталлятор — это DLL-библиотека, помогающая установить драйвер в систему.

После всех этих манипуляций с нашим драйвером неплохо бы сказать несколько слов об отладке драйверов.

6.4. Отладка драйверов

Я расскажу о классическом способе отладки драйверов — с использованием стандартного отладчика Microsoft, поставляемого в составе Microsoft DDK. Отладчик — это я, конечно, применила, слишком общее название. С Microsoft DDK поставляется несколько инструментов, так или иначе применяемых при отладке драйверов:

- KD — консольная утилита, используемая для отладки драйверов режима ядра;
- NTSD — консольная утилита, используемая для отладки драйверов пользовательского режима;
- CDB — разновидность утилиты NTSD. Применяется в основном для отладки консольных программ;
- WinDbg — отладчик с графическим интерфейсом, используемый для отладки драйверов как режима ядра, так и пользовательского режима.

Каждый из этих отладчиков имеет свои преимущества и свои ограничения. Так, NTSD не имеет возможности делать дамп памяти. А вот KD (так же как и WinDbg) имеет такую возможность. Также KD может отлаживать компьютеры разных архитектур (*x*86, Alpha (RISC), IA64), для чего предназначены разные версии этого инструмента.

Все наиболее нужные и полезные для "общения" с этими инструментами (и в обращении с отладкой вообще) термины вы сможете посмотреть в словаре терминов в *приложении 1*. А мы двигаемся дальше.

Прежде чем мы поговорим об использовании каждого из вышеперечисленных инструментов, сначала нужно посмотреть, как подготовить приложения к отладке. Для этого предназначены функции — отдельные для пользовательского режима и для режима ядра.

Для пользовательского режима предоставлены две функции:

- функция вывода отладочных сообщений (листинг 6.17);
- функция установки точек останова (breakpoints) (листинг 6.18).

Листинг 6.17. Функция `OutputDebugString`

```
VOID OutputDebugString(
   LPCTSTR lpOutputString
   );
```

Листинг 6.18. Функция `DebugBreak`

```
VOID DebugBreak(VOID);
```

Для режима ядра есть три вида функций:

- функции вывода отладочных сообщений (листинги 6.19 и 6.20);
- функции установки точек останова (breakpoints) (листинги 6.21—6.24);
- assert-функции (листинги 6.25 и 6.26).

Листинг 6.19. Функция `DbgPrint` выводит отладочные сообщения

```
ULONG
DbgPrint(
         PCHAR   Format,
         ...     [arguments]
);
```

Листинг 6.20. Функция `KdPrint` выводит отладочные сообщения (используется только в случае драйвера отладочной версии)

```
ULONG
KdPrint ( (
           PCHAR   Format,
           ...     [arguments]
) ) ;
```

Листинг 6.21. Функция `DbgBreakPoint` устанавливает точку останова

```
VOID
NTAPI
  DbgBreakPoint(
    VOID
  );
```

Листинг 6.22. Функция `DbgBreakPointWithStatus` устанавливает точку останова и посылает отладчику 32-битный код статуса

```
NTSYSAPI
VOID
NTAPI
  DbgBreakPointWithStatus(
    IN ULONG Status
  );
```

Листинг 6.23.Функция `KdBreakPoint` — то же, что `DbgBreakPoint` (только если драйвер — отладочная версия)

```
VOID
NTAPI
  KdBreakPoint(
    VOID
  );
```

Листинг 6.24. Функция `KdBreakPointWithStatus` — то же, что `DbgBreakPointWithStatus` (только если драйвер — отладочная версия)

```
NTSYSAPI
VOID
NTAPI
  KdBreakPointWithStatus(
    IN ULONG Status
  );
```

Листинг 6.25. Функция `ASSERT` — обычная assert-функция

```
VOID
  ASSERT(
    Expression
  );
```

Листинг 6.26. Функция ASSERTMSG — помимо основных assert-функций, позволяет послать дополнительное сообщение отладчику для вывода на экран

```
VOID
  ASSERTMSG(
    IN PCHAR Message,
    Expression
  );
```

* * *

Итак, я перечислила все главные функции, применяемые при отладке драйверов. Рассказывать о работе с графическими (консольными) утилитами и отладки или их установке, я думаю, не стоит — это очень просто и интуитивно понятно.

А теперь мы получили все необходимые знания для того, чтобы начать разбираться с основами написания более сложных драйверов для настоящих устройств.

Глава 7

Сложные драйверы для Windows

Мы уже получили необходимый набор знаний и навыков, достаточный для уверенного и грамотного написания простейших драйверов для ОС Windows. Теперь мы можем двигаться дальше и посмотреть, как же пишутся настоящие сложные драйверы для реальных устройств. В этой главе мы рассмотрим написание драйверов для принтера, дисплея и видеокарты и фильтр-драйвера для USB-камеры. Упор будет сделан на общую информацию и подробный разбор структуры драйвера. Если взять исходный код любого настоящего драйвера, скажем, для того же принтера, то при его комментировании придется объяснять огромное количество малоочевидных деталей, специфичных для этого устройства. Это уже нужно для профессионалов, а это выходит за рамки книги. Тем не менее, дать основные знания, необходимые для успешного продвижения по этому пути, мне кажется обязательным.

7.1. Драйвер для принтера

Приступим к написанию драйвера для принтера.

Сначала посмотрим, каковы компоненты принт-спулера Windows. Но прежде всего необходимо разобрать вообще архитектуру печати в Windows.

Самые "большие" компоненты этой архитектуры — это принтерный спулер и набор драйверов принтеров. Что такое спулер?

Спулер — это компонент архитектуры Windows, являющийся сервером печати, работающим с очередями печати.

Прежде чем перейти к обсуждению и написанию драйверов для принтеров, разберемся до конца со спулером (так мы и будем его далее называть, по-

скольку этот термин гораздо распространеннее и больше "прижился" в IT-кругах).

Каково строение спулера — его архитектура, выражаясь по-научному? В данном случае, раз мы выяснили, что спулер — это *сервер* печати, то ясно, что уместно говорить о клиент-серверной архитектуре.

К клиенту в данном случае относятся:

- приложение, которому нужны услуги печати или какие-либо, связанные с ними, которые может ему предоставить спулер; все запросы приложение отправляет к GDI (GDI — следующий компонент, нижестоящий);
- Winspool.drv — пользовательский интерфейс, предоставляемый спулером. Этот компонент находится еще ниже. Чтобы больше не говорить об этом, отмечу, что все компоненты в этом списке перечисляются от вышестоящих к нижестоящим.

В данном случае клиентские компоненты нас пока мало интересуют; перейдем к серверным компонентам:

- Spoolsv.exe — API-сервер спулера;
- Spoolss.dll — это так называемый "роутер" спулера. Он разбирает поступающие к нему запросы и определяет, к какому провайдеру их нужно переадресовать.

И, наконец, самое важное и интересное для нас — это провайдер печати.

Все компоненты (кроме пользовательского приложения и GDI) и составляют спулер. Клиент с сервером "общаются" с помощью RPC.

Что такое провайдер печати и что он делает? *Провайдер печати* — это компонент, работающий с определенными для него локальными и удаленными устройствами печати. Также через провайдер печати можно производить различные действия с очередями печатями. Сейчас мы поговорим о нем подробнее.

Провайдеры печати бывают следующих типов:

- провайдер локальной печати; файл — localspl.dll;
- провайдер сетевой печати; файл — win32spl.dll;
- провайдер печати Novell NetWare; файл — win32spl.dll;
- провайдер печати, работающий с HTTP; файл — inetpp.dll.

С сетевыми провайдерами мы разбираться в данной книге не будем; а поговорим о провайдере локальной печати.

Каковы вообще основные функции провайдера печати? Что он должен делать? Задач у него очень много, но если говорить коротко, то основное на-

значение провайдеров печати — управлять всеми ресурсами в системе, связанными с печатью:

- ❒ очередями печати;
- ❒ драйвером принтера;
- ❒ заданиями принтера;
- ❒ портами

и т. д.

Все функции, определенные провайдером печати, делятся на следующие:

- ❒ функции инициализации;
- ❒ функции управления драйвером принтера, очередями печати, заданиями принтера, портами, реестром и т. д.

Кроме того, есть еще функция `XcvData`, обеспечивающая связь между серверной и клиентской DLL-библиотеками монитора порта.

Локальный провайдер печати, помимо поддержки стандартного набора функций, также должен поддерживать:

- ❒ архитектуру драйвера принтера вместе с вызовами к DLL-библиотеке интерфейса локального принтера;
- ❒ архитектуру предоставляемого производителем процессора печати;
- ❒ архитектуру предоставляемого производителем монитора порта.

Ну, все, хватит о спулере. Перейдем собственно к драйверу принтера.

Существует несколько типов драйверов принтера для Windows:

- ❒ Microsoft Universal Printer Driver — универсальный драйвер принтера;
- ❒ Microsoft PostScript Printer Driver — драйвер для PostScript-принтера;
- ❒ Microsoft Plotter Driver — драйвер для плоттера.

В данном случае мы будет рассматривать создание универсального драйвера принтера. Начнем его писать, а по ходу дела буду давать необходимые объяснения.

Мы разобьем этот процесс на три компонента и будем реализовывать их отдельно:

- ❒ плагин для Microsoft Render;
- ❒ мини-драйвер принтера;
- ❒ монитор порта.

Мини-драйвер принтера отвечает за предоставление информации о принтере для механизма рендеринга. Рендер перехватывает задания принтера, формирует из них растровые строки, а затем уже передает их спулеру. Ин-

формация о принтере ему нужна для того, чтобы корректно обработать задания принтера. Мини-драйвер принтера создается при помощи утилиты Unitool, входящей в состав DDK. Что делает эта утилита? Благодаря ей данные GPC (General Printer Charecterization) для одного или нескольких схожих растровых принтеров будут определены и организованы в мини-драйвере.

Теперь что касается плагина для Microsoft Render. Существует зарегистрированная функция под названием `IPrintOemUni::FilterGraphics`, которая получает доступ к сформированным растровым строкам перед их отправкой спулеру. Это дает возможность модифицировать строки перед отправкой: зашифровать, сжать и т. д. Прототип функции `IPrintOemUni::FilterGraphics` представлен в листинге 7.1.

Листинг 7.1. Функция `IPrintOemUni::FilterGraphics`

```
STDMETHOD
  (FilterGraphics) (
    THIS_
    PDEVOBJ pdevobj,
    PBYTE pBuf,
    DWORD dwLen
    )PURE;
```

Здесь параметр `pdevobj` содержит указатель на структуру `DEVOBJ`, `pBuf` — адрес растрового буфера, содержащего данные, и `wLen` — размер буфера, заданного параметром `pBuf`. Собственно, функцией `IPrintOemUni::FilterGraphics` главным образом и реализуется плагин. Но, конечно, у него есть и много других функций.

Монитор порта (port monitor) — это часть архитектуры подсистемы печати Windows NT. Каждый монитор порта поддерживает стандартный набор API-функций. Вот функции, которые обязательно должны быть реализованы в мониторе порта:

- `InitializePrintMonitor`;
- `DllEntryPoint`;
- `OpenPort`;
- `OpenPortEx`;
- `ClosePort`;
- `StartDocPort`;

- `WritePort`;
- `ReadPort`;
- `EndDocPort`;
- `AddPort` (`AddPortEx`);
- `DeletePort`;
- `EnumPorts`;
- `ConfigurePort`;
- `SetPortTimeOuts`;
- `GetPrinterDataFromPort`.

Спулер вызывает эти функции по мере надобности. Их функциональность разъяснена далее.

Начнем с функции `InitializePrintMonitor` (листинг 7.2).

Листинг 7.2. Функция `InitializePrintMonitor`

```
LPMONITOREX
InitializePrintMonitor (LPWSTR pRegistryRoot); // определенный
// вызывающим указатель на строку, содержащую путь в реестре,
// по которому принт-монитор сохраняет нужные ему данные
```

Спулер вызывает эту функцию во время инициализации и получает от этой функции структуру, содержащую точки входа для остальных функций. У монитора порта есть только две точки входа. Одна из них находится в этой функции, а другая — в функции `DllEntryPoint` (листинг 7.3). Монитор порта экспортирует все функции в структуре, полученной спулером.

Листинг 7.3. Функция `DllEntryPoint`

```
BOOL WINAPI
DllEntryPoint (HINSTANCE hInst, DWORD fdwReason, LPVOID lpvReserved);
```

Эта функция вызывается спулером в момент загрузки им монитора порта посредством функции Win32 API `LoadLibrary`. В остальных случаях играет роль точки входа для спулера для загрузки DLL-библиотеки в память и больше ничего не делает.

Следующая — функция `OpenPort` (листинг 7.4).

Листинг 7.4. Функция `OpenPort`

```
BOOL (WINAPI *pfnOpenPort) (LPWSTR pName,     // указатель на строку
                                              // с именем порта
                            PHANDLE pHandle); // указатель на
      // расположение, где будет размещен полученный дескриптор порта
```

Спулер вызывает эту функцию в момент назначения порта принтеру. Эта функция возвращает дескриптор порта в поле `pName`. Спулер использует возвращенный функцией дескриптор в последующих вызовах монитора порта: `StartDocPort`, `WritePort`, `ReadPort` и `EndDocPort`. Спулер ожидает завершения функции `OpenPort` (успеха или неудачи) приемлемое время. Все процедуры инициализации, которые может иметь монитор порта, выполняются в этой функции.

Далее — функция `ClosePort` (листинг 7.5).

Листинг 7.5. Функция `ClosePort`

```
BOOL (WINAPI *pfnClosePort) (HANDLE hPort); // указатель на дескриптор
                                            // порта
```

Спулер обычно вызывает эту функцию, когда порту, указанному параметром `hPort`, больше не соответствует ни одного принтера.

Формат вызова функции `StartDocPort` представлен в листинге 7.6.

Листинг 7.6. Функция `StartDocPort`

```
BOOL (WINAPI *pfnStartDocPort)
(HANDLE hPort,         // дескриптор порта
 LPWSTR pPrinterName, // строка с именем принтера
 DWORD  JobId,         // идентификатор работы
 DWORD  Level,         // определяет тип структуры, указатель
                       // на которую находится в pDocInfo
 LPBYTE pDocInfo);     // указатель на структуру DOC_INFO_X,
                       // описывающую документ, который нужно напечатать
```

Спулер вызывает эту функцию, когда он готов к отсылке задания на принтер.

Формат вызова функции `WritePort` представлен в листинге 7.7.

Листинг 7.7. Функция `WritePort`

```
BOOL (WINAPI *pfnWritePort)
(HANDLE  hPort,       // дескриптор порта
 LPBYTE  pBuffer,     // указатель на буфер, содержащий данные для печати
 DWORD   cbBuf,       // размер pBuffer
 LPDWORD pcbWritten); // указатель на расположение, где будет размещено
                      // число байтов, успешно записанных в порт
```

Функция `WritePort` посылает данные, указанные в `pBuffer`, принтеру. Спулер вызывает эту функцию, если ему необходимо отослать полное задание на принтер. Спулер устанавливает размер блока в параметре `cbBuf`. Если от принтера нет отклика, `WritePort` будет ждать приемлемое время и, не получив ответа, вернет `FALSE`. Функция `WritePort` всегда должна проинициализировать `pcbWritten` нулем перед попыткой записи в порт. Если попытка записи в порт оказалась успешной, то в `pcbWritten` будет находиться число посланных байтов.

Формат вызова функции `ReadPort` представлен в листинге 7.8.

Листинг 7.8. Функция `ReadPort`

```
BOOL (WINAPI *pfnReadPort)
(HANDLE  hPort,     // дескриптор порта
 LPBYTE  pBuffer,   // указатель на буфер, куда будут записаны
                    // полученные данные
 DWORD   cbBuffer,  // размер pBuffer
 LPDWORD pcbRead);  // указатель на расположение, где будет размещено
                    // число успешно прочитанных байтов
```

Функция `ReadPort` поддерживает принтеры, обеспечивающие двунаправленный обмен информацией. Если принтер передаст какие-либо данные, то найти их можно будет в `pBuffer`. Если принтер ничего не передает, то `ReadPort` выждет положенное время, чтобы окончательно убедиться в отсутствии поступающих от принтера данных, и вернет `FALSE`. Функция `ReadPort` всегда должна проинициализировать `pcbRead` нулем перед попыткой принятия данных от принтера. Если чтение данных, принятых от принтера, оказалось успешным, то `pcbRead` будет содержать число переданных байтов.

Следующая — функция `EndDocPort` (листинг 7.9).

Листинг 7.9. Функция EndDocPort

```
BOOL (WINAPI *pfnEndDocPort)
(HANDLE hPort); // дескриптор порта
```

Принтер вызывает функцию EndDocPort после того, как задание завершено. Мониторы должны вызвать Win32-функцию SetJob для того, чтобы проинформировать спулер о завершении работы. Функция монитора порта EndDocPort должна вызвать SetJob с параметром dwCommand, установленным в JOB_CONTROL_SENT_TO_PRINTER. Когда работа принтера проходит через "языковый" монитор, спулер игнорирует любые оповещения, получаемые от монитора порта. Следовательно, монитор, который может определить реальное окончание работы принтера, должен задержать вызов SetJob до тех пор, пока принтер не уведомит об окончании работы. Для этой цели может использоваться функция EndDocPort. Language-монитор должен передать JOB_CONTROL_LAST_PAGE_EJECTED, когда он получает уведомление от принтера об окончании работы. Мониторам может понадобиться изменить это, если пользователь убирал или перезапускал задание. Для того чтобы определить происхождение этого события, нужно вызвать Win32-функцию GetJob и проверить, не установлен ли статус работы в JOB_STATUS_DELETING или JOB_STATUS_RESTART. Функция EndDocPort также должна освободить все ресурсы, выделенные функцией StartDocPort.

Далее — функция AddPort (листинг 7.10).

Листинг 7.10. Функция AddPort

```
BOOL (WINAPI *pfnAddPort)
(LPWSTR pName,    // указатель на строку (обязательно заканчивающуюся 0),
                  // содержащую имя сервера, к которому будет
                  // подсоединен порт;
                  // если 0 — то используется локальный порт
 HWND   hWnd,     // дескриптор родительского окна диалогового окна,
                  // в котором будет введено имя порта
 LPWSTR pMonitorName);    // указатель на строку (обязательно
                          // заканчивающуюся 0), содержащую имя монитора,
                          // ассоциированного с портом
```

AddPort создает порт и добавляет его в список портов, поддерживаемых в настоящий момент указанным монитором в окружении спулера. AddPort

разрешает интерактивное добавление портов. Монитор должен спросить пользователя об имени порта в диалоговом окне, ассоциированном с параметром `hWnd`. Функция `AddPort` должна проверить введенное имя порта путем вызова Win32-функции `EnumPorts`, которая проверяет, что в окружении спулера нет портов с идентичными именами. Монитор должен также удостовериться в том, что порт входит в число поддерживаемых им.

Формат вызова функции `DeletePort` представлен в листинге 7.11.

Листинг 7.11. Функция `DeletePort`

```
BOOL (WINAPI *pfnDeletePort)
(LPWSTR pName, // указатель на строку (обязательно заканчивающуюся 0),
               // содержащую имя сервера, у которого будет удален порт;
               // если 0 — то используется локальный порт
 HWND   hWnd,  // дескриптор родительского окна диалогового окна,
               // в котором будет введено имя порта
 LPWSTR pPortName);  // указатель на строку (обязательно заканчивающуюся
                     // 0), содержащую имя порта, который нужно удалить
```

Спулер вызывает эту функцию для удаления порта из окружения монитора. Монитор должен удалить указанный порт из своего состояния.

Формат вызова функции `EnumPorts` представлен в листинге 7.12.

Листинг 7.12. Функция `EnumPorts`

```
BOOL (WINAPI *pfnEnumPorts)
(LPWSTR  pName,      // указатель на строку, содержащую имя сервера,
                     // на котором будут перечисляться порты;
                     // если 0, то будет использоваться сервер,
                     // на котором выполняется серверная DLL-библиотека
 DWORD   Level,      // задает тип структуры буфера,
                     // указатель на который находится в pPorts
 LPBYTE  pPorts,     // указатель на буфер приема информации
                     // со структурой PORT_INFO_X, описывающей
                     // поддерживаемые порты принтера
 DWORD   cbBuf,      // размер буфера, указатель на который
                     // содержится в pPorts
```

```
 LPDWORD pcbNeeded,  // указатель на адрес
                     // размера буфера, необходимо для приема
                     // всей возвращенной информации
 LPDWORD pcbReturned);  // указатель на расположение количества
                        // перечисленных портов
```

Спулер вызывает `EnumPorts` для получения списка портов, содержащихся в мониторе. Во время своей инициализации спулер вызывает `EnumPorts` для всех установленных мониторов порта, чтобы создать список доступных портов.

Формат вызова функции `ConfigurePort` представлен в листинге 7.13.

Листинг 7.13. Функция `ConfigurePort`

```
BOOL (WINAPI *pfnConfigurePort)
(LPWSTR pName,  // указатель на строку, содержащую имя сервера,
                // на котором находится требуемый порт;
                // если 0, то будет использоваться локальный порт
 HWND   hWnd,   // дескриптор родительского диалогового окна,
                // в котором будет введена информация
 LPWSTR pPortName); // указатель на строку (обязательно заканчивающуюся
    // 0), содержащую имя порта, который нужно сконфигурировать
```

Спулер вызывает функцию `ConfigurePort` для выполнения конфигурирования порта. `ConfigurePort` может вывести диалоговое окно для получения от пользователя всей или только некоторой необходимой информации для конфигурирования принтера.

Формат вызова функции `SetPortTimeOuts` представлен в листинге 7.14.

Листинг 7.14. Функция `SetPortTimeOuts`

```
BOOL(WINAPI *pfnSetPortTimeOuts)
(HANDLE         hPort,     // дескриптор открытого порта, для которого
                           // нужно установить значения тайм-аута
 LPCOMMTIMEOUTS lpCTO,     // указатель на структуру COMMTIMEOUTS
 DWORD          reserved); // для будущего использования;
                           // значение должно быть 0
```

Устанавливает тайм-аут ожидания ответа порта. Необязательная функция.

Формат вызова функции `GetPrinterDataFromPort` представлен в листинге 7.15.

Листинг 7.15. Функция `GetPrinterDataFromPort`

```
BOOL (WINAPI *pfnGetPrinterDataFromPort)
(HANDLE  hPort,         // дескриптор порта
 DWORD   ControlID,     // IOCTL устройства. Если 0, то см. pValueName
 LPWSTR  pValueName,    // указатель на строку, определяющую информацию,
                        // которая будет запрошена;
                        // только если ControlID = 0
 LPWSTR  lpInBuffer,    // указатель на буфер, содержащий входные данные;
                        // только если ControlID не равен 0;
 DWORD   cbInBuffer,    // размер буфера lpInBuffer
 LPWSTR  lpOutBuffer,   // указатель на буфер, принимающий
                        // запрошенную информацию
 DWORD   cbOutBuffer,   // размер lpOutBuffer
 LPDWORD lpcbReturned ); // указатель на число байтов,
                          // содержащихся в lpOutBuffer
```

Получает данные принтера из порта. Необязательная функция.

Уф-ф, кажется все. Если у вас есть DDK (а он обязан у вас быть), то в каталоге ddk\src\print\localmon вы сможете найти пример монитора порта.

Итак, все вышеперечисленные функции мы реализовали. У нас есть примитивный драйвер принтера.

7.2. Драйвер для дисплея и драйвер для видеокарты

Сейчас мы разберем принципы написания минипорт-драйвера для дисплея и видеокарты с поддержкой многих мониторов. Повторяю, что все изложенное далее верно только для NT-систем, а в данном случае — для Windows XP и Windows Server 2003.

При написании минипорт-драйвера для видеокарт мы должны писать его для одной карты или для семейства видеокарт. В ситуации же с дисплеями

мы можем написать один минипорт-драйвер для любых мониторов, поддерживающих общий интерфейс. Например, минипорт-драйвер VGA подходит для любых мониторов, поддерживающих VGA, и т. д.

Для начала разберемся с драйвером для дисплея. Каковы компоненты процесса отображения в Windows NT? В пользовательском режиме находится приложение и среда Win32 (в которою также включен GDI 32 пользовательского режима).

Прежде чем идти дальше, поговорим о том, что такое GDI.

GDI (Graphics Driver Interface) — это графический интерфейс между графическими (прошу прощения за тавтологию) драйверами Windows NT и приложениями. Приложения посылают свои запросы пользовательскому GDI, который пересылает их GDI режима ядра, а тот, в свою очередь, перенаправляет эти запросы графическим драйверам.

GDI взаимодействует с графическими драйверами через набор функций DDI (Device Driver Interface). Эти функции легко отличить от любых других — все они имеют префикс Drv. Графические же драйверы взаимодействуют с GDI режима ядра с помощью набора функций GDI.

Посмотрим, как GDI взаимодействует с драйвером. Для того чтобы успешно взаимодействовать с GDI, драйверу необходимо экспортировать всего одну функцию — `DrvEnableDriver` (листинг 7.16).

Листинг 7.16. Функция `DrvEnableDriver`

```
BOOL DrvEnableDriver(
  IN ULONG  iEngineVersion, // параметр, означающий версию запущенного
                            // в данный момент экземпляра GDI
  IN ULONG  cj,             // содержит размер структуры DRVENABLEDATA
                            // (в байтах)
  OUT DRVENABLEDATA  *pded  // указатель на структуру DRVENABLEDATA
);
```

Функция `DrvEnableDriver` возвращает `TRUE`, если нужный драйвер был включен. В противном случае — `FALSE`. Понятно, что эта функция должна присутствовать во всех графических драйверах.

Взглянем на структуру `DRVENABLEDATA` (листинг 7.17).

Листинг 7.17. Структура DRVENABLEDATA

```
typedef struct tagDRVENABLEDATA {
   ULONG iDriverVersion; // указывает версию DDI,
                         // для которой драйвер создается
   ULONG c;              // указывает количество DRVFN-структур в буфере,
                         // указатель на который хранится в pdrvfn
   DRVFN *pdrvfn;        // указатель на буфер, в котором хранится
                         // массив DRVFN-структур
}  DRVENABLEDATA, *PDRVENABLEDATA;
```

Теперь, похоже, нужно посмотреть на структуру DRVFN (листинг 7.18).

Листинг 7.18. Структура DRVFN

```
typedef struct _DRVFN {
   ULONG iFunc; // индекс функции, определяющий предоставляемую
                // функцию DDI
   PFN   pfn;   // указывает адрес определенной драйвером функции DDI,
                // соотнесенной с индексом в iFunc
} DRVFN, *PDRVFN;
```

Вообще, эта структура используется графическими драйверами для предоставления GDI указателей на DDI-функции.

Рассматриваем компоненты процесса отображения дальше. В режиме ядра находятся GDI "ядерного" режима, взаимодействующий с менеджером ввода/вывода и драйвером дисплея, который взаимодействует с собственно графическим адаптером, а тот, в свою очередь, "общается" с видеоминипортом, видеопортом и менеджером ввода/вывода. Круг замкнулся.

Драйвер дисплея — это DLL-библиотека режима ядра, чья главная роль — рендеринг. Когда приложение вызывает функцию Win32 с независимым от устройства графическим запросом, то интерфейс GDI интерпретирует эти запросы в вызовы к драйверу устройства. Драйвер устройства, в свою очередь, интерпретирует эти инструкции в команды к устройству для рисования графики на экране. По умолчанию GDI обрабатывает запросы рисования, как запросы стандартного растрового формата. Но драйвер дисплея может перехватить и обработать эти запросы так, как необходимо в конкретном случае.

Для специальных, критических по времени, запросов драйвер дисплея может непосредственно обратиться к регистрам видеоустройства.

Теперь о минипорт-драйверах видеокарты. Минипорт-драйвер видеокарты главным образом обрабатывает запросы, которые должны взаимодействовать с другими компонентами ядра Windows NT. Например, такие запросы, как инициализация устройства. Минипорт-драйвер видеокарты должен управлять всеми ресурсами (например, ресурсами памяти), общими для минипорт-драйвера видеокарты и драйвера дисплея. Система не гарантирует, что ресурсы, назначенные драйверу дисплея, всегда будут доступны для минипорт-драйвера видеокарты. Минипорт-драйвер видеокарты также должен выполнять следующую работу:

- интерактивная установка режимов видеокарты;
- минимизация "железных" зависимостей в драйвере дисплея

и т. д.

Теперь посмотрим, какими методами драйвер дисплея может взаимодействовать с видеокартой.

Во-первых, при помощи посылки минипорт-драйверу графического адаптера IOCTL-запросов, а во-вторых, путем прямого чтения/записи памяти видеокарты или регистров устройства. Разберем оба способа поподробнее.

Вкратце: при первом способе драйвер дисплея вызывает функцию `EngDeviceIoControl` с кодом IOCTL для отсылки синхронного запроса к минипорт-драйверу видеокарты. GDI использует один буфер для ввода и вывода при обработке запроса к подсистеме ввода/вывода. Подсистема ввода/вывода перенаправляет запрос к порту видеокарты, который обрабатывает запрос вместе с минипорт-драйвером видеокарты. Необходимо отметить, что посылка IOCTL-запросов минипорт-драйверу для реализации критических по времени функций может снизить общую производительность системы.

Использование второго способа — прямого доступа к памяти видеокарты и/или регистрам устройства — помогает радикально уменьшить объем кода драйвера. В этом случае мы можем предоставить только те функции, которые драйвер дисплея может выполнить быстрее, чем GDI, а обработку всех остальных функций возложить на GDI.

Теории, думаю, пока достаточно. Теперь приведу список главных функций, которые должны быть реализованы в драйвере дисплея (листинг 7.19).

Листинг 7.19. Функция `DrvEnableDriver`

```
BOOL DrvEnableDriver(
  IN ULONG  iEngineVersion, // параметр, означающий версию запущенного
                            // в данный момент экземпляра GDI
  IN ULONG  cj,             // содержит размер структуры DRVENABLEDATA
                            // (в байтах)
  OUT DRVENABLEDATA *pded   // указатель на структуру DRVENABLEDATA
);
```

Это первая функция драйвера дисплея, которая вызывается операционной системой. При вызове этой функции драйвер сообщает GDI информацию о версии драйвера и точках входа функций, предоставляемых драйвером.

Теперь рассмотрим функцию `DrvDisableDriver` (листинг 7.20).

Листинг 7.20. Функция `DrvDisableDriver`

```
VOID DrvDisableDriver(VOID);
```

Эта последняя функция, вызываемая в минипорт-драйвере. В этой функции драйвер должен освободить все занятые ресурсы, чтобы получить возможность быть безопасно выгруженным.

Рассмотрим функцию `DrvEnablePDEV` (листинг 7.21).

Листинг 7.21. Функция `DrvEnablePDEV`

```
DHPDEV DrvEnablePDEV(
  IN DEVMODEW *pdm,            // указатель на структуру DEVMODEW
                               // драйвера, содержащую данные
  IN LPWSTR   pwszLogAddress,  // драйвер дисплея должен проигнорировать
                               // этот параметр
  IN ULONG    cPat,            // драйвер дисплея должен проигнорировать
                               // этот параметр
  OUT HSURF   *phsurfPatterns, // драйвер дисплея должен проигнорировать
                               // этот параметр
  IN ULONG    cjCaps,          // размер буфера, указатель на который
                               // находится в pdevcaps
```

```
  OUT ULONG    *pdevcaps,        // указатель на структуру GDIINFO
  IN ULONG     cjDevInfo,        // размер структуры, указатель на которую
                                 // находится в pdi
  OUT DEVINFO *pdi,              // указатель на структуру DEVINFO
  IN HDEV      hdev,             // определенный GDI дескриптор устройства
  IN LPWSTR    pwszDeviceName,   // указатель на строку (которая
                                 // обязательно должна заканчиваться 0),
                                 // содержащую понятное пользователю
                                 // имя устройства
  IN HANDLE   hDriver            // идентифицирует драйвер режима ядра,
                                 // поддерживающий устройства
);
```

GDI вызывает эту функцию, когда ему нужно получить информацию об устройстве, обслуживаемом драйвером. При получении этого вызова драйвер должен выполнить следующие основные действия:

1. Удостовериться, что структура `GDIINFO` заполнена в соответствии с описанными в документации, прилагаемой к DDK, правилами.
2. Заполнить структуру `DEVINFO` в соответствии с описанными в документации, прилагаемой к DDK, правилами.
3. Сохранить полученный GDI дескриптор в закрытой переменной `DEVMOD`.
4. Сохранить имя устройства (`pwszDeviceName`) и дескриптор устройства (`hDriver`).
5. Вернуть `PDEV` нашей структуре описания устройства.

Рассмотрим функцию `DrvDisablePDEV` (листинг 7.22).

Листинг 7.22. Функция `DrvDisablePDEV`

```
VOID DrvDisablePDEV(
  IN DHPDEV  dhpdev  // указатель на структуру PDEV, определяющую
                     // физическое устройство, которое нужно отключить
);
```

Эта функция удаляет объект `PDEV`, созданный функцией `DrvEnablePDEV`. Она также должна освободить все ресурсы, занятые функцией `DrvEnablePDEV`.

Далее — функция `DrvEnableSurface` (листинг 7.23).

Листинг 7.23. Функция `DrvEnableSurface`

```
HSURF DrvEnableSurface(
  IN DHPDEV  dhpdev  // указатель на структуру PDEV, определяющую
                     // физическое устройство, для которого
                     // создается "поверхность"
);
```

Функция вызывается для создания так называемой "поверхности", используемой для операций рисования. Здесь могут быть два метода, используемых для достижения целей (листинги 7.24 и 7.25).

Листинг 7.24. Метод `EngCreateDeviceSurface` — поверхность, управляемая драйвером

```
HSURF EngCreateDeviceSurface(
  DHSURF dhsurf,       // указывает "поверхность", которая будет
                       // управляться устройством
  SIZEL  sizl,         // пиксельный размер созданной поверхности
  ULONG  iFormatCompat // определяет совместимый машинный формат
                       // для создаваемой "поверхности" устройства
);
```

Листинг 7.25. Метод `EngModifySurface` — поверхность, управляемая устройством (необходима лишь для некоторых операций рисования)

```
BOOL EngModifySurface(
  IN HSURF  hsurf,      // определяет поверхность, которая будет изменена
  IN HDEV   hdev,       // определяет устройство,
                        // с которым будет ассоциирована "поверхность"
  IN FLONG  flHooks,    // набор флагов, контролирующих функции,
                        // которые драйвер может перехватить каждый раз,
                        // когда GDI "рисует" на "поверхности"
  IN FLONG  flSurface,  // набор флагов, определяющих атрибуты
                        // "поверхности"
  IN DHSURF dhsurf,     // определяет поверхность драйвера
  OUT VOID* pvScan0,    // указатель на виртуальный адрес начала
                        // точечного рисунка
```

```
  IN LONG   lDelta,     // указатель на виртуальный адрес шага
                        // точечного рисунка
  IN VOID*  pvReserved  // зарезервировано для будущего использования;
                        // должно быть установлено в 0;
);
```

Рассмотрим функцию `DrvDisableSurface` (листинг 7.26).

Листинг 7.26. Функция `DrvDisableSurface`

```
VOID DrvDisableSurface(
  IN DHPDEV  dhpdev   // указатель на структуру PDEV, определяющую
                      // физическое устройство, "поверхность" которого
                      // нужно очистить
);
```

Освобождает ресурсы, занятые функцией `DrvEnableSurface`. Здесь необходим вызов, по крайней мере, `EngDeleteSurface`, для освобождения всех занятых функцией `DrvEnableSurface` ресурсов.

Рассмотрим функцию `DrvGetModes` (листинг 7.27).

Листинг 7.27. Функция `DrvGetModes`

```
ULONG DrvGetModes(
  IN HANDLE    hDriver,  // указатель на драйвер, для которого
                         // должны быть перечислены режимы
  IN ULONG     cjSize,   // размер буфера, указатель на который
                         // находится в *pdm
  OUT DEVMODEW *pdm      // хранит указатель на буфер, содержаший
                         // структуру DEVMODEW для заполнения;
                         // другой вариант — 0;
);
```

Эта функция используется для перечисления всех режимов, поддерживаемых устройством (в данном случае — дисплеем).

Рассмотрим функцию `DrvAssertMode` (листинг 7.28).

Листинг 7.28. Функция `DrvAssertMode`

```
BOOL DrvAssertMode(
  IN DHPDEV  dhpdev,  // определяет указатель на структуру PDEV,
                      // описывающую "железный" режим, который
                      // должен быть установлен, если bEnable задан
  IN BOOL    bEnable  // определяет режим, в который будет установлено
                      // аппаратное обеспечение
);
```

Для этой функции необходим параметр `bEnable`. Если он установлен в `TRUE`, то драйвер дисплея должен проверить параметр `dhpdev`, является ли он правильным для этого устройства.

Рассмотрим функцию `DrvCompletePDEV` (листинг 7.29).

Листинг 7.29. Функция `DrvCompletePDEV`

```
VOID DrvCompletePDEV(
  IN DHPDEV  dhpdev, // идентифицирует физическое устройство
                     // по его дескриптору, который возвращает GDI
                     // после вызова DrvEnablePDEV
  IN HDEV    hdev    // определяет физическое устройство,
                     // которое было установлено
);
```

Сохраняет дескриптор, который может потребоваться в будущем для вызова GDI-функций. Этот дескриптор используется функциями вида `Eng`*`XXX`*`()`.

Драйверу дисплея могут потребоваться функции, перехватывающие DDI, для записи на дисплей.

Рассмотрим сначала функцию `DrvCopyBits` (листинг 7.30).

Листинг 7.30. Функция `DrvCopyBits`

```
BOOL DrvCopyBits(
  IN SURFOBJ  *psoDst,  // указатель на принимающую "поверхность"
                        // в операции копирования
  IN SURFOBJ  *psoSrc,  // указатель на исходную "поверхность"
                        // в операции копирования
```

```
  IN CLIPOBJ  *pco,     // указатель на CLIPOBJ, определяющую область
                        // обрезки на принимающей "поверхности"
  IN XLATEOBJ *pxlo,    // указатель на XLATEOBJ, определяющую
                        // конвертацию цветовых индексов между исходной
                        // и принимающей "поверхностями"
  IN RECTL    *prclDst, // указатель на структуру RECTL,
                        // определяющую область модификации
  IN POINTL   *pptlSrc  // указатель на структуру POINTL, определяющую
                  // верхний левый угол исходной области-прямоугольника
);
```

Эта функция необходима для записи на дисплей в случае, если карта памяти видеокарты недоступна для дисплея.

Рассмотрим функцию `DrvBitBlt` (листинг 7.31).

Листинг 7.31. Функция `DrvBitBlt`

```
BOOL DrvBitBlt(
  IN SURFOBJ  *psoDst, // указатель на структуру SURFOBJ, описывающую
                       // поверхность для рисования
  IN SURFOBJ  *psoSrc, // указатель на структуру SURFOBJ, описывающую
                       // источник для блоковых операций перемещения;
                       // если это нужно, то указывается в rop4
  IN SURFOBJ  *psoMask,  // указатель на структуру SURFOBJ, описывающую
                         // поверхность, которая будет использована
                         // в качестве маски для rop4
  IN CLIPOBJ  *pco,      // указатель на структуру CLIPOBJ,
                         // ограничивающую область модификации
  IN XLATEOBJ  *pxlo,    // указатель на XLATEOBJ, определяющую
                         // конвертацию цветовых индексов между исходной
                         // и принимающей "поверхностями";
                         // если 0, то конвертация не требуется
  IN RECTL  *prclDst,    // указатель на структуру RECTL,
                         // определяющую область модификации
  IN POINTL  *pptlSrc,   // указатель на структуру POINTL, определяющую
                         // верхний левый угол исходной
                         // области-прямоугольника
```

```
  IN POINTL  *pptlMask,  // указатель на структуру POINTL, которая
                         // определяет, какой пиксел маски совпадает
                         // с левым верхним углом исходной
                         // области-прямоугольника;
                         // если *psoMask = 0, то этот параметр
                         // игнорируется
  IN BRUSHOBJ  *pbo,     // указатель на объект кисти,
                         // определяющий шаблон для блоковой передачи
  IN POINTL  *pptlBrush, // указатель на структуру POINTL, которая
                         // определяет источник кисти
                         // у принимающей "поверхности"
  IN ROP4  rop4          // специфицирует растровую операцию, которая
                         // определяет как маска, шаблон, источник
                         // и пикселы "приемника" комбинируются для
                         // единой записи на принимающую "поверхность"
);
```

Эта функция может быть необходима для записи DIB-данных на VGA-устройство.

Перейдем к минипорт-драйверу видеокарты.

Заголовочные файлы, которые должны (или могут) понадобиться в минипорт-драйвере видеокарты, следующие:

- ntddvdeo.h — содержит IOCTL-коды и структуры, посылаемые минипорт-драйверу в VRP-пакетах;
- miniport.h — содержит базовые типы, константы и структуры для минипорта видеокарты;
- video.h — содержит прототипы функций `VideoPortXxx`, `SvgaHwIoPortXxx` и др.;
- videoagp.h — содержит прототипы функций для AGP-драйверов;
- tvout.h — содержит определение структуры `VIDEOPARAMETERS`;
- devioctl.h — содержит константы и макросы, используемые для определения IOCTL-кодов;
- dderror.h — содержит коды ошибок, используемые драйвером.

Теперь поговорим о функциях, необходимых в минипорт-драйвере видеокарты.

Начнем с функции `DriverEntry` (листинг 7.32).

Листинг 7.32. Функция `DriverEntry`

```
ULONG DriverEntry(
  IN PVOID  Context1,   // Указатель на контекстное значение, с которым
    // минипорт-драйвер должен вызвать VideoPortInitialize.
    // Это контекстное значение идентифицирует объект драйвера,
    // созданный системой для минипорт-драйвера.
  IN PVOID  Context2    // Указатель на контекстное значение, с которым
    // минипорт-драйвер должен вызвать VideoPortInitialize.
    // Это контекстное значение идентифицирует путь в реестре
    // для минипорт-драйвера.
);
```

Данная функция инициализирует переменные расширения драйвера. При необходимости выделяет все нужные ресурсы.

Рассмотрим функцию `HwVidFindAdapter` (листинг 7.33).

Листинг 7.33. Функция `HwVidFindAdapter`

```
VP_STATUS (*PVIDEO_HW_FIND_ADAPTER)(
  IN PVOID  HwDeviceExtension, // указатель на область хранения драйвера
  IN PVOID  HwContext,         // должно быть проигнорировано
  IN PWSTR  ArgumentString,    // указатель на ASCII-строку,
                               // оканчивающуюся 0; строка определяется
                               // пользователем; может быть 0;
  IN OUT PVIDEO_PORT_CONFIG_INFO  ConfigInfo, // указатель на структуру
      // VIDEO_PORT_CONFIG_INFO, содержащую конфигурационную
      // информацию адаптера, определяемую шиной
  OUT PUCHAR  Again            // должно быть проигнорировано
);
```

Определяет адаптеры, присутствующие в системе, с помощью PnP (Plag-and-Play) и IO-менеджеров (Input/Output, ввод/вывод). Для каждого найденного устройства получает дескриптор/информацию об устройстве. Если устройство имеет какие-либо ресурсы, то получает их.

Рассмотрим функцию `HwVidGetPowerState` (листинг 7.34).

Листинг 7.34. Функция `HwVidGetPowerState`

```
VP_STATUS (*PVIDEO_HW_POWER_GET)(
  PVOID  HwDeviceExtension,    // указатель на область хранения драйвера
  ULONG  HwId,          // указатель на 32-битное число, идентифицирующее
    // устройство, для которого мини-порт будет запрашивать информацию
  PVIDEO_POWER_MANAGEMENT  VideoPowerControl   // указатель на структуру
    // VIDEO_POWER_MANAGEMENT, определяющую состояние питания,
    // для которого будет запрошено наличие поддержки
);
```

Если устройство поддерживает запросы информации о состоянии питания, то функция возвращает эту информацию. В противном случае возвращает ошибку.

Рассмотрим функцию `HwVidGetVideoChildDescriptor` (листинг 7.35).

Листинг 7.35. Функция `HwVidGetVideoChildDescriptor`

```
ULONG (*PVIDEO_HW_GET_CHILD_DESCRIPTOR)(
  IN PVOID  HwDeviceExtension, // указатель на область хранения драйвера
  IN PVIDEO_CHILD_ENUM_INFO  ChildEnumInfo,    // указатель на структуру
    // VIDEO_CHILD_ENUM_INFO, определяющую устройство,
    // которое будет описано
  OUT PVIDEO_CHILD_TYPE  VideoChildType,    // указатель на расположение,
    // куда минипорт-драйвер вернет тип дочернего устройства,
    // которое будет описано
  OUT PUCHAR  pChildDescriptor,  // указатель на буфер, в который
    // минипорт-драйвер вернет данные, идентифицирующие устройство
  OUT PULONG  UId, // указатель на буфер, куда минипорт-драйвер поместит
    // возвращенный уникальный 32-битный идентификатор устройства
  OUT PULONG  pUnused            // не используется; должно быть 0;
);
```

Если мы поддерживаем любую конфигурацию монитора, то можем предоставить здесь какую-либо информацию. Иначе мы можем сообщить о типе дочернего устройства просто как "Монитор".

Рассмотрим функцию `HwVidInitialize` (листинг 7.36).

Листинг 7.36. Функция `HwVidInitialize`

```
BOOLEAN (*PVIDEO_HW_INITIALIZE)(
  PVOID  HwDeviceExtension // указатель на область хранения драйвера
);
```

Эта функция выполняет любые специфичные для устройства процедуры инициализации.

Рассмотрим функцию `HwVidSetPowerState` (листинг 7.37).

Листинг 7.37. Функция `HwVidSetPowerState`

```
VP_STATUS (*PVIDEO_HW_POWER_SET)(
  PVOID  HwDeviceExtension,   // указатель на область хранения драйвера
  ULONG  HwId,        // указатель на 32-битное число, идентифицирующее
    // устройство, для которого будет установлено состояние питания
  PVIDEO_POWER_MANAGEMENT  VideoPowerControl  // указатель на структуру
    // VIDEO_POWER_MANAGEMENT, определяющую состояние питания,
    // которое необходимо установить
);
```

Устанавливает состояние питания монитора или адаптера (см. описание функции `GetPowerState`).

Рассмотрим функцию `HwVidStartIo` (листинг 7.38).

Листинг 7.38. Функция `HwVidStartIo`

```
BOOLEAN (*PVIDEO_HW_START_IO)(
  PVOID  HwDeviceExtension,    // указатель на область хранения драйвера
  PVIDEO_REQUEST_PACKET  RequestPacket     // указатель на структуру
    // VIDEO_REQUEST_PACKET, содержащую все параметры,
    // первоначально переданные в EngDeviceIoControl
);
```

Когда адаптер уже проинициализирован, драйвер видеопорта и драйвер дисплея вызывают функцию `StartIo` для выполнения всех специфицированных для дисплея операций.

Итак, реализация вышеприведенных функций позволит нам получить примитивный минипорт-драйвер видеокарты.

7.3. Фильтр-драйвер для USB-камеры

Наиболее важные для этой темы понятия:

- пины — представляют каналы ввода/вывода (I/O) устройства;
- фильтры — все ясно.

А теперь приступлю собственно к предмету изложения.

Фильтр-драйвер USB-камеры обычно предназначен для обработки поступающих потоковых видеоданных. Все операции над данными (как, например, их фильтрация) производятся без буферизации (не влияя на общую производительность системы).

USB-камера начинает захватывать поступающие видео- и звуковые данные после включения. FDO (Functional Device Object) узнает о том, что камера стала доступной посредством уведомления Plug-and-Play (которое является callback-функцией). FDO обязан инициализировать структуры заголовка потока для хранения в них поступающих данных.

Для аудио- и видеоданных определены отдельные потоки. Каждый поток характеризуется конкретными параметрами: пропускная способность, максимальный размер пакета, тип передачи данных, их формат и т. д.

Клиентское приложение получает уведомления о входящих потоках с помощью функций-"уведомителей" о событиях, определенных самими приложениями. FDO "генерирует" событие всякий раз, когда происходит чтение потока. Это происходит асинхронно, что повышает скорость работы драйвера.

FiDO (Filter Device Object) может использоваться для расширения возможностей FDO. Этот фильтр-драйвер реализован как Upper filter driver, который будет помогать модифицировать нам входящие и исходящие потоки данных. Менеджер ввода/вывода генерирует IRP, которые перехватываются FiDO.

Большинство функций в потоковом фильтр-драйвере почти такие же, как в обычном фильтр-драйвере. Потоковый фильтр-драйвер должен поддерживать предопределенный набор IOCTL. Разберем подробнее функции потокового фильтр-драйвера.

Каковы функции, инициализирующие устройство и открывающие пин? Первым делом — `SRB_INITIALIZE_DEVICE`. Эта функция инициализирует

библиотеку USBCAMD, а также выполняет конфигурирование и инициализацию устройства. Функция SRB_UNINITIALIZE_DEVICE выполняет освобождение ресурсов. Функция SRB_OPEN_STREAM, как явствует из названия, открывает поток, а SRB_CLOSE_STREAM — закрывает.

Другие важные функции — SRB_GET_STREAM_INFO, позволяющая получить информацию о потоке, SRB_SET_DATAFORMAT, устанавливающая используемый формат данных. Также есть две функции SRB_CHANGE_POWER_STATE, изменяющие состояние питания из включенного в выключенное (и, соответственно, наоборот).

Потоковый фильтр-драйвер обязан создать подпрограммы для передачи *всех* IRP, даже если он и не собирается всех их обрабатывать. Одним словом, фильтр-драйвер должен поддерживать все IRP, которые поддерживает конечное устройство. Если фильтр-драйвер "забудет" хотя бы об одном из этих IRP, то он (этот IRP) будет завершен с ошибкой STATUS_INVALID_DEVICE_REQUEST. Все вышесказанное касается функции DriverEntry. Функция AddDevice в данном случае внимания не заслуживает, поэтому перейдем к функциям, специфичным для потокового фильтр-драйвера.

Большинство работ по обработке входящего потока реализовано с помощью управляющих кодов. Чтобы перехватить данные потока и изменить их, фильтр-драйвер должен обработать все IOCTL. Данные потока могут быть получены путем использования IOCTL-кодов IOCTL_KS_READ_STREAM и IOCTL_KS_WRITE_STREAM.

Два главных IOCTL, с помощью которых драйвер может получить доступ к данным потока, — IOCTL_KS_READ_STREAM и IOCTL_KS_WRITE_STREAM, для чтения и записи соответственно. Эти запросы содержат список StreamHeader (заголовок потока), который дополнительно заполняется настоящими актуальными данными. Если поток стандартен, то используется структура KSSTREAM_HEADER.

Структура KSSTREAM_HEADER описывает время представления и опции, относящиеся к буферу данных и потоку. Эта структура содержит указатель на виртуальный адрес настоящих данных KSSTREAM_HEADER.Data.

Теперь об обработке IOCTL_KS_PROPERTY.

IOCTL IOCTL_KS_PROPERTY, как вы наверняка помните, предоставляет клиенту возможность получать информацию о потоках и наборах свойств. Классовый драйвер потока использует функцию KsPropertyHandler, которая обрабатывает все запросы свойств, произошедшие через IOCTL_KS_PROPERTY. Фильтр-драйвер не может вызывать эту функцию непосредственно и обрабатывать этот IOCTL. Для этой цели WDM Streaming предоставляет функцию KsDispatchSpecificProperty.

Функция `KsDispatchSpecificProperty` посылает свойство назначенному обработчику. Эта функция предполагает, что вызывающая программа сначала передала IRP функции `KsPropertyHandler`.

Большинство `IOCTL_KS_PROPERTY`-запросов в функции обратного вызова `KsPropertyHandler` (ее тип — `PFNKSHANDLER`). Эта функция принимает три параметра: собственно сам IRP, указатель на структуру `KSIDENTIFIER` и указатель на буфер данных. Второй параметр содержит информацию о типе и цели запроса.

Структура `KSIDENTIFIER` содержит в себе три переменные: первая из них, `KsIdentifier.Set` (тип `GUID`), показывает тип поддерживаемого набора свойств. Значение второй, `KsIdentifier.Id` (тип `ULONG`), зависит от первой переменной. Третья переменная, `KsIdentifier.Flags`, указывает, что нужно делать с информацией свойства — установить или прочитать.

Свойства потока и состояние соединения могут быть получены с помощью наборов свойств `KSPROPSETID_Pin` и `KSPROPSETID_Connection`. Набор свойств потока может дать такую информацию, как тип потока, тип коммуникации, интерфейсы и т. д. С помощью набора свойств соединения можно узнать о состоянии потока, формате данных (если он менялся) и т. п. Тип сделанного запроса можно узнать с посредством `KsIdentifier.Id`.

Это вкратце о потоковых драйверах вообще. А фильтр-драйвер специально для USB-камеры характеризируется специальным набором структур, функций обратного вызова и сервисов. Я приведу здесь лишь наиболее характерные из них.

Рассмотрим сначала структуру `USBCAMD_DEVICE_DATA` (листинг 7.39).

Листинг 7.39. Структура `USBCAMD_DEVICE_DATA`

```
typedef struct _USBCAMD_DEVICE_DATA
{
  ULONG                              Sig;  // для будущего использования
  PCAM_INITIALIZE_ROUTINE            CamInitialize;
  PCAM_INITIALIZE_ROUTINE            CamUnInitialize
  PCAM_PROCESS_PACKET_ROUTINE        CamProcessUSBPacket;
  PCAM_NEW_FRAME_ROUTINE             CamNewVideoFrame;
  PCAM_PROCESS_RAW_FRAME_ROUTINE     CamProcessRawVideoFrame;
  PCAM_START_CAPTURE_ROUTINE         CamStartCapture;
  PCAM_STOP_CAPTURE_ROUTINE          CamStopCapture;
  PCAM_CONFIGURE_ROUTINE             CamConfigure;
```

```
  PCAM_STATE_ROUTINE                CamSaveState;
  PCAM_STATE_ROUTINE                CamRestoreState;
  PCAM_ALLOCATE_BW_ROUTINE          CamAllocateBandwidth;
  PCAM_FREE_BW_ROUTINE              CamFreeBandwidth;
} USBCAMD_DEVICE_DATA, *PUSBCAMD_DEVICE_DATA;
```

Назначения параметров интуитивно понятны — не будем тратить на них время. Эта структура используется для определения точек входа для функций обратного вызова мини-драйвера камеры.

Теперь рассмотрим структуру `USBCAMD_INTERFACE` (листинг 7.40).

Листинг 7.40. Структура `USBCAMD_INTERFACE`

```
typedef struct {
  INTERFACE                         Interface;
  PFNUSBCAMD_WaitOnDeviceEvent      USBCAMD_WaitOnDeviceEvent;
  PFNUSBCAMD_BulkReadWrite          USBCAMD_BulkReadWrite;
  PFNUSBCAMD_SetVideoFormat         USBCAMD_SetVideoFormat;
  PFNUSBCAMD_SetIsoPipeState        USBCAMD_SetIsoPipeState;
  PFNUSBCAMD_CancelBulkReadWrite    USBCAMD_CancelBulkReadWrite;
} USBCAMD_INTERFACE, *PUSBCAMD_INTERFACE;
```

Здесь назначение параметров также интуитивно понятно. Структура определяет набор функций обратного вызова, соотнесенных с интерфейсом шины USB.

Теперь немного о функциях обратного вызова (листинги 7.41—7.43).

Листинг 7.41. Функция `CamInitialize`

```
NTSTATUS
CamInitialize(
  PDEVICE_OBJECT BusDeviceObject,    // указатель на объект устройства
                       // мини-драйвера камеры, созданный USB-хабом
  PVOID DeviceContext  // указатель на контекст устройства
                       // мини-драйвера камеры
);
```

Эта функция инициализирует устройство (в данном случае — камеру).

Листинг 7.42. Функция `CamStartCapture`

```
NTSTATUS
CamStartCapture(
  PDEVICE_OBJECT BusDeviceObject,   // указатель на объект устройства
                         // мини-драйвера камеры, созданный USB-хабом
  PVOID DeviceContext    // указатель на контекст устройства
                         // мини-драйвера камеры
);
```

Эта функция подготавливает устройство к передаче данных.

Листинг 7.43. Функция `CamNewVideoFrame`

```
VOID
CamNewVideoFrame(
  PVOID DeviceContext, // указатель на контекст устройства
                       // мини-драйвера камеры
  PVOID FrameContext   // указатель на контекст фрейма
                       // мини-драйвера камеры
);
```

Эта функция инициализирует контекст нового видеофрейма.

Полный код драйвера USBCAMD вы сможете найти в каталоге примеров каталога DDK в wdm\videocap\usbcamd.

* * *

Итак, мы разобрались с основами и принципами написания более сложных драйверов (сделали некие наметки, если можно так выразиться) и увидели, что, в общем-то, это довольно просто.

Итак, первая логическая часть книги подошла к концу. Мы получили огромный набор базовых знаний — об архитектуре Windows, WDM, технологиях, применяемых в разработке драйверов, научились писать простейшие драйверы и получили представление о том, как пишутся настоящие, сложные драйверы. Теперь мы обладаем всеми необходимыми знаниями и навыками для того, чтобы приступить к разговору о будущем (фактически уже настоящем) — о новейшей ОС Vista, о новейшей драйверной технологии WDF и т. д.

Глава 8

Мультипроцессорная парадигма программирования

Среди прочего в этой книге я рассматриваю написание многопоточных драйверов — драйверов для многопроцессорных систем. Пока еще можно считать, что эта область программирования практически в самом начале пути своего развития. Тема эта достаточно сложная и, однако, актуальная; ее актуальность возрастает вместе с увеличением популярности систем, для которых предназначены многопоточные драйверы. Тем не менее, сейчас мы уже можем получить всю информацию, необходимую для дальнейшего изучения этой области и адекватного восприятия всех возникающих в ней инноваций.

Начнем мы, как обычно, с основ и рассмотрим, что такое мультипроцессинг, многопоточность и т. д. — все базовые понятия этой области, которые знать необходимо.

Итак, мультипроцессинг как таковой.

8.1. Мультипроцессинг

Для начала, для того чтобы уяснить себе, откуда вообще возникло это понятие, рассмотрим такую вещь, как способы работы с данными и инструкциями в компьютере. Естественно, мы будем рассматривать эти способы с общей — концептуальной, так сказать, — точки зрения, особенно не затрагивая конкретные технические детали и подробности.

Если взять классическую модель процессорных архитектур, созданную в 1966 году Майклом Флинном (я делаю такую оговорку потому, что на данный момент это уже давно не единственная, хоть и пока еще традиционная

модель процессорных архитектур), то мы сможем выделить четыре способа работы с данными и инструкцией в компьютере.

Из четырех способов, фактически и составляющих главным образом эту модель, лишь один относится к традиционной немультипроцессорной архитектуре компьютера. Остальные три относятся непосредственно к теме нашего текущего разговора — мультипроцессорности, многопоточности и т. д. Разберем все четыре.

Прежде всего, каково главное понятие архитектуры, созданной Флинном? Эта архитектура базируется на ключевом понятии *потока*. Что понимается под понятием "поток" в данной конкретной архитектуре? У Флинна поток — это определенная последовательность данных, команд и т. д., обрабатываемая процессором. Легко понять отсюда, что все четыре способа отличаются главным образом числом потоков — потоков данных и команд. Итак.

- SISD — single instruction (stream)/single data (stream). Полностью последовательный способ работы с данными и инструкциями. Как видно из названия, в этом случае мы имеем один поток инструкций и один поток данных. Пример такого рода компьютеров — PDP-11. Вообще, такой способ работы используют все машины, относящиеся к классическому типу (типу фон Неймана).
- SIMD — single instruction (stream)/multiple data (stream). Здесь мы уже имеем, как видно, по-прежнему один поток команд, но для данных уже есть несколько потоков. Еще одно значительное изменение структуры этого способа — это наличие в потоке команд таких вещей, как векторы. Что они дают? Использование векторов позволяет производить одну и ту же операцию над несколькими элементами данных — элементами вектора. Пример такой машины — CRAY-1.
- MISD — multiple instruction (stream)/single data (stream) — достаточно необычный способ работы. Здесь мы видим один поток данных и несколько потоков инструкций. Что может реализовать такую модель? В принципе, из описания этого способа следует, что есть несколько процессоров, при этом, однако, обрабатывающих один поток данных. "Официально" класс машин, использующих такой способ работы, считается "пустым".
- MIMD — multiple instructions (stream)/multiple instructions (stream) — и, наконец, мы получили несколько потоков инструкций и несколько потоков данных. Здесь мы, наконец, подошли вплотную непосредственно к теме нашего обсуждения. Итак: несколько процессоров (ядер), объединенных в одно, но работающих каждый над своими потоками данных и инструкций.

Почему я отдельно выделила модель Майкла Флинна? Дело в том, что эта модель на данный момент уже далеко не единственная. Создано большое количество других самых разных моделей (например, проект многоклеточных процессоров), многие из которых занимают свою определенную нишу в этой области. Но пока что модель Майкла Флинна все равно остается традиционной и общепринятой (пусть порой с некоторыми дополнениями и говорками), а рассмотрение других моделей не входит в рамки нашего обсуждения (по крайне мере, в настоящей книге).

Итак. Сначала мы поговорим о мультипроцессорных концепциях с точки зрения "железа", а затем — с точки зрения ОС, системного и прикладного ПО.

Мы не будем вдаваться в долгие рассуждения насчет преимуществ AMD-процессоров перед Intel-процессорами (или наоборот), а просто примем за аксиому, что Intel-процессоры — это лидеры; прежде всего, по распространенности (что и являлось главным критерием при выборе темы обсуждения).

Так что начнем обсуждать многопроцессорные технологии компании Intel. Сразу скажу, что, независимо от того, что мы будем обсуждать в этой главе — спецификацию ли многопроцессорных систем, или многопроцессорную архитектуру — *везде* (за исключением случаев, когда это будет оговорено особо) речь идет о компании Intel, ее разработках и процессорах.

8.2. Многопроцессорность и многоядерность от компании Intel: спецификация MPS

В данном разделе речь сначала пойдет о спецификации многопроцессорных систем, созданной компанией Intel.

MPS (Multiprocessor Specification) как таковая появилась на свет в октябре 1993 года. Это был pre-release версии 1.1. Сама версия 1.1 появилась в ноябре 1994 года. Впоследствии — до января 1995 года — компания Intel выпускала только исправления и дополнения этой своей спецификации. В январе же 1995 года свет увидела MPS версии 1.4.

Что же такое — эта MPS? В общем-то, и сама эта спецификация создавалась не с нуля. Она, в сущности, является дополнением к уже существующему стандарту проектирования DOS-совместимых систем.

Какие основные компоненты системы определяет MP-спецификация (как иначе называется MPS)? MP-спецификация определяет в системе MP-BIOS (что самое главное) набор структур данных конфигурации MP. Оба компо-

нента связаны самым "тесным" образом — структуры данных MP создают MP-BIOS.

Каковы минимальные технические требования к аппаратным средствам, установленные в спецификации MP?

- ❒ Минимум один процессор с набором команд, совместимым с Intel 486 и Pentium.
- ❒ Минимум один APIC-контроллер (на процессорах Pentium).
- ❒ Подсистема кэша и подсистема общей памяти. Обе они должны быть прозрачны для программ.
- ❒ Компоненты PC/AT-платформ, также обязательно видимые для программ.

Что такое APIC-контроллер? APIC-контроллер (Advanced Programmable Interrupt Controller) — это контроллер с распределенной архитектурой. В чем выражается эта его "распределенность"? APIC-контроллер распределяет функции управления прерываниями между двумя компонентами: блоком ввода/вывода (I/O-блоком) и локальным блоком. Естественно, этим блокам необходимо обмениваться информацией. Они делают это посредством ICC-шины (Interrupt Controller Communication) — шины коммуникаций контроллера прерываний.

Все эти вышеперечисленные аппаратные компоненты должны, кроме того, предоставлять (как того требует MP-спецификация) возможность выполнения следующих требований:

- ❒ память:
  - конфигурация системной памяти;
  - кэшируемость и доступность физической памяти для процессоров;
  - управление памятью;
  - сортировка записей в памяти;
  - отображение памяти APIC;
- ❒ прерывания:
  - управление прерываниями;
  - режимы прерывания; их должно быть три:
    - ◊ режим PIC; обходит APIC; в этом режиме система функционирует как однопроцессорная;
    - ◊ режим виртуальной линии; использует APIC в качестве виртуальной линии; все остальное — идентично — с PIC;
    - ◊ режим симметричного ввода/вывода; использует все возможности многопроцессорной системы;

- распределение системы прерываний (это касается локальных APIC-блоков);

❒ таймеры интервалов;

❒ поддержка перезагрузки.

Поговорим о MP-BIOS. Этот тип BIOS для многопроцессорных систем, помимо обычных для BIOS функций, таких как:

❒ тестирование компонентов системы;

❒ построение таблицы конфигурации для использования операционной системой;

❒ инициализация процессора и приведение системы в нужное состояние;

❒ предоставление сервисов времени выполнения, работающих с устройствами

обязан делать еще следующее:

❒ предоставлять операционной системе конфигурационную информацию обо всех имеющихся в системе процессорах и других мультипроцессинговых компонентах в системе;

❒ проинициализировать все процессоры и привести все мультипроцессинговые компоненты в нужное состояние.

Приступим к рассмотрению структуры MP-системы с точки зрения MP-спецификации. Ее компоненты:

❒ системные процессоры;

❒ APIC-контроллеры;

❒ системная память;

❒ шины расширения ввода/вывода.

Здесь, я думаю, все понятно. Об APIC-контроллерах мы уже говорили ранее.

Перейдем к обсуждению структур данных конфигурации MP.

Для чего вообще нужны эти структуры? Для предоставления операционной системе информации о конфигурации MP. Принимать эту информацию операционная система может двумя способами:

❒ минимальный способ (или способ-минимум) — этот способ позволяет задать конфигурацию MP с помощью выбора одного из существующих стандартных наборов значений параметров аппаратного обеспечения;

❒ максимальный способ (или способ-максимум) — этот, соответственно, предоставляет большие и гибкие возможности конфигурирования.

Итак, какие структуры данных используются при описании конфигурации MP-системы? Их две:

- структура указателя переходов;
- таблица конфигурации MP (необязательна).

Поговорим подробнее о каждой из этих структур.

В структуре указателя переходов содержатся физические адреса указателей на таблицу конфигурации MP и прочие информационные структуры MP. Вообще, наличие структуры указателя переходов однозначно сигнализирует о соответствии системы спецификации MP.

Операционная система ищет эту структуру в нескольких местах, поэтому она должна быть расположена, как минимум, в одном из них:

- первый килобайт EBDA (Extended BIOS Data Area, расширенная область данных BIOS);
- последний килобайт базовой системной памяти;
- в BIOS ROM (адреса между 0F0000h и 0FFFFFh).

Размер этой структуры — 16 байтов. В табл. 8.1 описываются поля этой структуры.

Таблица 8.1. *Поля структуры указателя переходов*

Название поля	Смещение	Длина (в битах)	Описание
SIGNATURE	0	32	ASCII-строка ("_MP_"), используемая как ключ поиска для нахождения структуры указателя
PHYSICAL ADDRESS POINTER	4	32	Адрес начала таблицы конфигурации MP. Если таковой нет, то вся длина поля будет "забита" нулями
LENGTH	8	8	Длина структуры указателя в так называемых параграфах. Каждый параграф равен 16 байтам; следовательно, для этой структуры значение этого поля будет равно 01h и соответствовать одному параграфу
SPEC_REV	9	8	Номер поддерживаемой версии MPS. Может быть равен 01h (для версии MPS 1.1) и 04h (понятно, для версии MPS 1.4)
CHECKSUM	10	8	Контрольная сумма полной структуры

Таблица 8.1 *(окончание)*

Название поля	Смещение	Длина (в битах)	Описание
MP FEATURE INFORMATION BYTE 1	11	8	Указывает тип системной конфигурации MP. Значение "0" указывает на наличие таблицы конфигурирования MP. Ненулевое же значение указывает на то, что стандартная конфигурация MP предоставляется системой
MP FEATURE INFORMATION BYTE 2	12:0 12:7	7 1	Биты с 0 по 6 зарезервированы для будущего использования. Седьмой бит — IMCRP. Если этот бит установлен, следовательно, IMCRP имеется и используется PIC-режим; в противном случае — режим виртуальной линии
MP FEATURE INFORMATION BYTE 3-5	13	24	Зарезервировано для будущего использования и поэтому должно быть равно "0"

Перейдем к таблице конфигурации MP. В ней содержится информация о процессорах, прерываниях, APIC-контроллерах и шинах.

Она состоит из двух частей: базовой секции (части) и дополнительной секции (части). Базовая часть состоит из полей, целиком обеспечивающих обратную совместимость с предыдущими версиями MPS. Дополнительная часть содержит, понятно, дополнительные поля. Кроме того, у этой таблицы есть заголовок (header).

Эта структура может находиться в следующих местах:

- первый килобайт EBDA (Extended BIOS Data Area, расширенная область данных BIOS);
- последний килобайт базовой системной памяти;
- в пространстве памяти BIOS только для чтения (read-only) (адреса между 0E0000h и 0FFFFFh);
- вершина системной физической памяти.

И снова — таблички полей частей структуры (табл. 8.2—8.4).

Таблица 8.2. *Поля заголовка таблицы конфигурации MP*

Название поля	Смещение	Размер (в битах)	Описание
SIGNATURE	0	32	Назначение поля аналогично таковому в структуре указателя переходов. ASCII-строка — "PCMP"

Таблица 8.2 *(окончание)*

Название поля	Смещение	Размер (в битах)	Описание
BASE TABLE LENGTH	4	16	Длина базовой секции таблицы конфигурации MP (включая и сам заголовок)
SPEC_REV	6	8	Аналогично таковому в структуре указателя переходов
CHECKSUM	7	8	Контрольная сумма базовой части таблицы
OEM_ID	8	64	Строка, определяющая производителя системного аппаратного обеспечения
PRODUCT_ID	16	96	Строка, определяющая класс (family), к которому относится продукт
OEM TABLE POINTER	28	32	Физический адрес указателя на определяемую производителем необязательную таблицу конфигурации. Если таковой не имеется, значение поля — "0"
OEM TABLE SIZE	32	16	Размер таблицы конфигурации, определяемой производителем. Если таблицы не имеется, значение поля, понятно, "0"
ENTRY COUNT	34	16	Количество полей в базовой части таблицы конфигурации MP
ADDRESS OF LOCAL APIC	36	32	Базовый адрес, посредством которого каждый конкретный процессор получает доступ к своему локальному APIC-контроллеру
EXTENDED TABLE LENGTH	40	16	Размер дополнительной части таблицы. Если ее нет, значение поля — "0"
EXTENDED TABLE CHECKSUM	42	8	Контрольная сумма дополнительной части таблицы конфигурации. Если ее нет, значение поля — "0"

Таблица 8.3. *Типы полей в базовой части таблицы конфигурации MP*

Назначение типа поля	Код типа поля	Размер (в байтах)	Описание
Процессор	0	20	Указывает процессор; этих полей столько же, сколько процессоров в системе
Шина	1	8	Указывает шину; этих полей столько же, сколько шин

Таблица 8.3 *(окончание)*

Назначение типа поля	Код типа поля	Размер (в байтах)	Описание
I/O APIC (ввод/вывод APIC-контроллера)	2	8	По одному полю на каждый
Назначение прерывания ввода/вывода	3	8	По одному на каждый источник шинных прерываний
Назначение локальных прерываний	4	8	По одному на каждый источник системных прерываний

Таблица 8.4. *Типы полей в дополнительной части таблицы конфигурации MP*

Название поля	Код типа поля	Размер (в байтах)	Описание
`SYSTEM ADDRESS SPACE MAPPING`	128	20	Поле для объявления видимой системной памяти или же пространства ввода/вывода шины
`BUS HIERARCHY DESCRIPTOR`	128	20	Поле для описания взаимосвязей шины ввода/вывода
`COMPATIBILITY BUS ADDRESS SPACE MODIFIER`	130	8	Поле для описания предопределенных интервалов адресов, используемых для изменения памяти или видимого пространства ввода/вывода на шине (это нужно для поддержания ISA-совместимости)

Ранее я уже упоминала о так называемых "дефолтных" (стандартных) конфигурациях системы MP. Настало время поговорить о них подробнее.

Итак, "дефолтная" конфигурация системы MP — это некий набор уже установленных параметров, который, при необходимости, можно использовать в "готовом виде". Возможность использования таких вот стандартных конфигураций (будем называть их далее так) есть далеко не на любых системах. Для того чтобы мы могли работать со стандартными конфигурационными наборами, наша система должна соответствовать следующим минимальным требованиям:

- ❒ в системе должны быть минимум два процессора;
- ❒ система должна поддерживать работу с несколькими процессорами;

- все имеющиеся в системе процессоры должны быть совместимы со стандартным набором команд Intel-процессоров;
- система должна поддерживать как PIC-режим, так и режим виртуальной линии; при этом каждый из них может быть использован как стандартный режим работы с прерываниями;
- локальные APIC-контроллеры должны быть расположены по адресу 0FEE0_0000h; а APIC ввода/вывода (I/O APIC) должны быть расположены по адресу 0FEC0_0000h;
- все ID локальных APIC-контроллеров должны быть назначены самими устройствами; при этом их ID должны идти последовательно друг за другом (отсчитывая от 0).

Вообще стандартных конфигураций MP может быть 255 штук. Но на данный момент (в текущей спецификации MP) создано всего 7. А все остальные "свободные места" зарезервированы для использования в будущем.

Понятно, что каждая конфигурация имеет свой уникальный ID, по которому может быть однозначно идентифицирована. Все они поддерживают два процессора и различные шины (а также и их сочетания).

Итак, более или менее спецификацию MPS рассмотрели. Все-таки книга не об этом; поэтому не будем слишком уж задерживаться на ней; эта глава ведь всего лишь вводная.

Начнем разбираться с процессором Itanium 2®. Itanium в очень многом похож на Itanium (только хуже). Но так как на данный момент Itanium 2 значительно актуальнее и прогрессивнее своего предшественника, то и рассматривать мы будем его намного пристальнее.

8.3. Процессоры Intel Itanium 2

Процессоры Intel Itanium 2 являются двухъядерными. В данном случае это означает, что процессоры этого поколения включают в себя два полнофункциональных 64-битных ядра. Кроме того, процессоры этой серии используют технологию EPIC. Вот о ней стоит поговорить поподробнее.

EPIC (Explicitly Parallel Instruction Computing) — новая архитектура, ключевой особенностью которой является явный параллелизм на уровне команд. Эта архитектура позволяет ПО напрямую обращаться к процессору, предоставляя при этом возможность параллельных вычислений. EPIC, выполняя распараллеливание еще на этапе компиляции, позволяет радикально повысить производительность обработки программного кода.

EPIC — архитектура, разработанная в 90-х годах прошлого века в университете Иллинойса. Первоначально разработка имела кодовое название Impact.

Начиная с разработки основных теоретических "постулатов" самой архитектуры и заканчивая разработкой инструментальных средств для нее, EPIC приобрела тот вид, какой она имеет сейчас.

Каковы главные особенности и признаки архитектуры EPIC?

- Как мы уже говорили, поддержка параллелизма еще при компиляции.
- Организация регистров в стек.
- Большой регистровый файл.
- Наличие предикатных регистров.
- Поддержка команд, выполняемых предикатно.
- Особая поддержка программной конвейеризации.

У архитектуры EPIC еще много интересных возможностей (связанных, главным образом, с работой с компилятором, программным кодом и командами), но все их мы здесь перечислять не будем.

Сначала немного поговорим о некоторых программных особенностях Intel Itanium 2, а затем уже — о программировании под него.

Что дает процессору использование EPIC-архитектуры? Наличие всех имеющихся у этой архитектуры преимуществ дает возможность Intel Itanium 2 обеспечивать теснейшую связь между аппаратным и программным обеспечением. В Intel Itanium 2 интерфейс между АО и ПО спроектирован таким образом, чтобы ПО имело возможность получать всю доступную на время компиляции информацию, а также быстро и эффективно предоставлять эту информацию АО.

Эффективная работа технологии параллелизма на уровне команд (эта технология называется ILP) в большой степени поддерживается в Intel Itanium 2 конвейерным механизмом шириной в шесть и глубиной в 8 этапов, работающим на частотах 900 МГц/1 ГГц.

Вообще-то, говоря об Intel Itanium, необходимо разобраться, главным образом, с архитектурой IA-64, тем более что EPIC — ее составляющая.

Первый процессор — Merced, созданный с применением этой архитектуры, был представлен в конце 1999 года. Технология до сих пор живет и развивается.

С технологией IA-64 появилось много новых возможностей, в нее включенных:

- длинные слова команд (LIW — long instructions words);
- устранение ветвлений (BE — branch elimination);
- предикаты команд (IP — instruction predication);
- предварительная загрузка команд (SL — speculative loading)

и т. д.

В IA-64 существуют два режима преобразования (декодирования) команд — новый VLIW и старый CISC (для совместимости). В данном случае нас интересует именно VLIW; о нем мы и поговорим.

VLIW (Very Long Instruction Word) определяет такую архитектуру процессоров с несколькими вычислительными устройствами, которая характеризуется тем, что несколько потоков инструкций выполняются параллельно.

В процессорах, использующих VLIW, распараллеливание осуществляется еще на этапе компиляции. При этом происходит явное указание того, какое именно устройство будет выполнять эту команду.

* * *

В следующей главе мы рассмотрим программирование многопоточных и 64-битных драйверов для процессоров Intel Itanium 2.

Глава 9

Написание 64-битных драйверов и драйверов для многопроцессорных систем

9.1. Написание 64-битных драйверов

9.1.1. Необходимые замечания

В предыдущей главе мы рассмотрели архитектуру и принципы работы многопроцессорных систем от Intel, а также разобрали архитектуру и особенности многопроцессорной парадигмы программирования.

В этой главе мы обсудим вопросы, непосредственно относящиеся к главной теме книги: это написание 64-битных драйверов для многопроцессорных систем под Windows.

Прежде всего, для каких операционных систем мы сможем позиционировать наши драйверы? Итак, для следующих 64-битных версий операционных систем семейства Windows:

- Microsoft Windows XP;
- Microsoft Windows Server 2003;
- Microsoft Windows Vista;
- Microsoft Windows Longhorn (кодовое имя для серверной ОС нового поколения).

Сначала рассмотрим все не совсем зависящие от программиста требования, которым должны удовлетворять ОС и устройства, для которых пишутся драйверы:

- устройства должны поддерживать технологию Plug and Play и технологию управления питанием;

- устройства должны поддерживать 64-битную адресацию памяти;
- системы, основанные на Itanium®, должны поддерживать 64-битную версию ACPI-таблицы.

Это, пожалуй, главные требования.

Теперь о программных требованиях.

Для программирования драйверов вам, конечно же, понадобится установленный WDK (Widows Driver Kit). Для того чтобы успешно начать писать 64-битные драйверы, вам необходимо проследовать по следующему пути: **Start | All Programs | Windows Driver Kits | WDK_Version | Build Environment | Operating System | Build Environment** (Пуск | Все программы | Windows Driver Kits | WDK_Version | Build Environment | Operating System | Build Environment).

В появившемся окне вы можете выбрать окружение, соответствующее вашим нуждам — IA-64 соответствует архитектуре Intel Itanium, а x64 — архитектуре x-64-based.

Итак, приступим.

9.1.2. Требования к драйверу, соглашение о вызовах и обзор главных изменений

Прежде всего, поговорим об ограничениях и требованиях, предъявляемых теперь к 64-битному драйверу:

- драйвер не может модифицировать код ядра во время выполнения;
- драйвер не может модифицировать такие таблицы, как IDT и GDT;
- драйвер не может модифицировать недокументированные структуры ядра;
- драйвер не может создавать и использовать свой собственный стек.

Это главные правила. Вообще-то, глубинных и не очень ограничений намного больше — за информацией предлагаю обратиться к соответствующим документам в MSDN.

Теперь поговорим о соглашении о вызовах. Вкратце рассмотрим calling convention (соглашение о вызовах; в дальнейшем — CC) 64-битных драйверов.

Изменения и дополнения коснулись самых разных аспектов CC в 64-битных драйверах.

Существуют три типа CC: STDCALL, FASTCAL и CDECL. Подробнее прочитать о них вы сможете в словаре терминов в конце книги.

Ближе всего CC 64-битных драйверов к FASTCALL. Главные отличия от последнего — это 64-битная адресация и наличие шестнадцати 64-битных регистров.

Что с типами? Биты, байты, слова и двойные слова остались прежними. Теперь у нас есть четвертные и восьмеричные слова — к сожалению, для этих типов данных еще не сложилось устоявшегося русскоязычного аналога. В английской номенклатуре это — Quadword — размером 64 бит, и Octalword — размером 128 байт. В языке C этим типам данных соответствуют:

- `_int64` (Quadword) — `int`;
- `unsigned _int64` (Quadword) — беззнаковый `int`;
- `double` (Quadword) — `FP64`;
- `struct __m64` (Quadword) — `__m64`;
- `struct __m128` (Octalword) — `__m128`.

Сначала указывается тип данных в языке C, в скобках — размер по новой номенклатуре, справа — какой скалярный тип данных всему этому соответствует.

Всевозможные агрегирования и объединения мы здесь рассматривать не будем — это не столь интересно.

Появилось много новых регистров, каждый из которых (при работе с функциями) предназначен для своей цели. Примеры:

- `RAX` — регистр возвращаемого значения;
- `RSP` — указатель стека;
- `RCX` — первый `int`-аргумент;
- `RDX` — второй `int`-аргумент.

Посмотрим на некоторые составляющие самого CC, а точнее, процесса его обработки.

Рассмотрим механизм обработки параметров.

1. Вызывающая функция передает четыре первых `int`-параметра (в порядке слева направо) в регистры `RCX`, `RDX`, `R8` и `R9` (регистры также перечислены слева направо).
2. Если вызывающая функция хочет передать какие-либо еще аргументы, то все они помещаются уже в стек.
3. Если вызывающей функции нужно передать какие-либо параметры типов с плавающей запятой или двойной точности, то они помещаются в регистры `XMM0`—`XMM3` (параметры с первого по четвертый соответственно) с

размещением дополнительной информации в регистрах для размещения int-параметров (RCX и т. д.).

4. Вызывающая функция никогда не передает массивы, строки и параметры типа __m128 прямо, а лишь передает ссылки на уже выделенную ею память. Структуры и объединения размером от 8 до 32 (а также 64-битные и __m64) передаются как целочисленные параметры того же размера.
5. Внутренние функции, которые не используют стек и не вызывают других функций, могут по желанию использовать другие регистры для передачи каких-либо своих параметров.
6. Вызывающая функция, если это нужно, должна сохранить "теневую" копию всех используемых регистров.

Большинство возвращаемых значений, "умещающихся" в 64 бита (__m64 входит в эту группу), передаются через регистр RAX — как уже отмечалось выше. Другие типы — __m128, __m128i, __m128d, float и double передаются через XMM0. Если же тип выходит за пределы 64 битов, то передается указатель на участок памяти, где он находится. Пользовательские типы могут быть длиной в 1, 2, 4, 8, 16, 32 и 64 бита.

Об СС представление получили.

Что касается поддержки 32-битного ввода/вывода в 64-битных драйверах, то подсистема WOW 64 позволяет 32-битным драйверам запускаться под 64-битной Windows. Этот механизм достаточно интересен, но разбирать мы его здесь не будем — это для нас сейчас не главное. Отмечу, однако, что эта подсистема работает только для приложений; для драйверов эта система не работает.

Лучше посмотрим, каковы новшества именно в программировании в 64-битных окружениях — компилятор, совместимость кода и т. д.

Компиляторы 64-битных систем должны выполнять несколько дополнительных функций — обеспечивать поддержку ANSI/ISO совместимости с языком C++, а кроме того, их компоновщики обязаны поддерживать оптимизацию так называемых "дальних переходов". Последняя опция позволяет компоновщику успешно работать с программами, регионы которых превышаю 16 Мбайт. Для включения этой возможности используется опция компилятора /opt:lbr.

Теперь о некоторых изменениях и дополнительных требованиях, предъявляемых к разработчику/языку при программировании на C и C++. Их тоже предостаточно; о необходимости поддержки совместимости с C++ уже говорилось. Стандартный размер паковки компоновщика теперь — 16 Мбайт. Кроме того, например, везде в коде, где не указан размер size_t и time_t, его теперь необходимо указать равным __int64 и т. д. Таких мелочей очень

много, тупо перечислять их нет смысла; опять же отсылаю к соответствующим недавно созданным и постоянно обновляемым документам.

Появились новые атрибуты, макросы и ключевые слова, используемые при программировании для 64-битных платформ.

Написана новая 64-битная библиотека времени выполнения для языка C.

А каковы ограничения 64-битного компилятора? Рассмотрим некоторые:

- отныне удалена опция `/clr` компилятора; компиляторы 64-битных платформ генерируют *только* низкоуровневый, "сырой" (native) код;
- больше не поддерживаются проверки безопасности, ранее исполняемые при включенной опции `/GS`;
- ключевое слово `_asm` больше не поддерживается; если вы хотите использовать ассемблерный код, его необходимо вынести в отдельный файл или же использовать встроенные функции.

Итак, "пробег" по разнообразным деталям завершен. Теперь перейдем уже, наконец, к достаточно большой теме — портированию 32-битных драйверов на 64-битную платформу и присущим этому процессу особенностям.

9.1.3. Портирование 32-битных драйверов на 64-битную платформу

Цель и концепция нового стиля программирования в 64-разрядной версии Windows — максимально унифицировать процесс разработки приложений одновременно и под 32-битную, и под 64-битную версии Windows. В более "радужном" варианте такой принцип предполагается обеспечить и для написания драйверов. Определенные шаги в этих направлениях уже сделаны.

Каковы изменения?

Очень многие профессиональные C-программисты привыкли к тому, что целочисленный, `long`-типы и указатели — одного размера (32 бита). Такой ситуации пришел конец — в новом 64-битном окружении эти типы *не* одинакового размера. Указатели теперь приобрели длину 64 бита. Понятно, что это необходимо, т. к. возникла задача адресации объемов памяти больших 16 Тбайт. А изменять размеры стандартных типов данных пока что нет никакой необходимости.

Еще одна деталь, которую стоит знать и помнить — в 64-битной ОС Windows не работает больше механизм автоматического устранения ошибок выравнивания памяти (в режиме ядра); поэтому перед переносом своего 32-битного драйвера на 64-битную платформу вы должны исправить все такие ошибки в его коде.

Теперь — новые типы данных. Они делятся на три группы: целочисленные с фиксированной точностью, целочисленные с точностью указателя и целочисленные со специальной точностью. Все эти типы "выведены" из стандартных типов C — `int` и `double` — а посему, новые типы данных прекрасно поддерживаются. Всем разработчикам предлагается уже сейчас работать с новыми типами данных и проверять работу написанного кода на обеих платформах — 32-битной и 64-битной.

Новые типы данных чрезвычайно полезны для устойчивости, поэтому переход весьма полезен и оправдан. К сожалению, этот переход не очень-то прост: необходимо просмотреть весь ваш код на предмет небезопасных использований указателей и т. д.

Как обеспечить поддержку DMA в 64-битной Windows? Для этого нужно, во-первых, проверить значение системной переменной `Mm64BitPhysicalAddress`, показывающей наличие поддержки 64-битной DMA; во-вторых, установить значение поля `Dma64BitAddresses` структуры описания устройства `DEVICE_DESCRIPTION`; в-третьих, использовать структуру `PHYSICAL_ADDRESS` для подсчета физических адресов — это главные требования. Стоит помнить, что если ваш код в 32-битном окружении нормально работал с DMA-функциями, то никаких проблем с его работой в 64-битном окружении быть не должно.

Итак, в завершение всего вышесказанного так называемая краткая памятка разработчику, портирующему/пишущему драйверы на 64-битной платформе:

- используйте новые безопасные типы данных;
- не забывайте о флагах формата в функциях `printf` и `wsprintf` (главным образом, модификатор `%p`);
- будьте внимательны при выполнении любых операций с пространствами памяти, знаковыми и беззнаковыми типами данных и 16-разрядными константами;
- обращайте внимание на использование операции `NOT` и будьте внимательны при вычислении размеров буферов;
- помните, что в 64-битной Windows 0xFFFFFFFF не равно −1! Пожалуй, наиболее часто встречающая ошибка;
- будьте осторожны с полиморфизмом и указателями структур.

Ну что ж, это главное.

> ***Примечание***
>
> Эта глава написана и построена в стиле "памятки" разработчику, в данном случае — разработчику, портирующему 32-битные драйверы на 64-битную платформу. Этот текст, конечно же, ни в коем случае не претендует на полноту

(помните, что самую полную информацию вы можете найти в MSDN), а лишь призван дать представление о типичных задачах, встающих перед разработчиком, портирующим драйверы с одной платформы на другую, и показать, что этот процесс портирования не так уж и сложен в данном случае.

9.2. Написание драйверов для многопроцессорных систем

Как известно, многопроцессорные системы подразумевают многопоточность. Драйверы для таких систем пишутся по другим принципам, главный из которых называется thread-safe. Проще говоря, любой многопоточный драйвер (будем называть его так) обязан корректно и безопасно работать с потоками — устанавливать приоритеты, блокировки и т. д. Вот об этом мы сейчас и поговорим.

Тема эта очень сложная и обширная. Мы рассмотрим только главные концепции и методы этой темы.

В каких случаях может понадобиться дополнительная работа с блокировками и прочим? В случае использования каких-либо общих данных, которые могут быть изменены во время работы с ними, а также при работе с набором каких-либо операций, которые могут быть выполнены отдельно.

Приведем пример, в котором использование блокировок необходимо. Представим, что у нас есть многопроцессорная система и код, создающий переменную и присваивающий ей какое-либо значение, а затем каким-то образом работающий с ней. Также предположим, что у нас есть и другой код, который в какой-либо момент времени читает значение этой переменной и использует его, скажем, для вывода баланса банковского счета.

Ясно, что если код № 2 обратится к переменной в тот момент, когда код № 1 уже изменил ее значение в памяти, но еще не записал его, то произойдет конфликт — код № 1 и код № 2 будут работать с разными версиями этих переменных, что может привести к непредсказуемым результатам. Ясно, что в данном случае во время обращения к переменной код № 1 должен установить блокировку, завершить все операции с переменной, а затем только снять блокировку.

Существует множество методов и концепций синхронизации, т. е. принципа использования общих данных разными "клиентами", каждый из которых, к тому же, может менять в какой-либо отрезок времени эти данные. Все их мы рассматривать не будем, а лучше рассмотрим тот "смешанный" механизм синхронизации, который используется в Windows.

Механизм синхронизации в Windows состоит из 7 компонентов.

Первый компонент — спин-блокировки, которые уже рассматривались на страницах этой книги. Эти самые спин-блокировки предоставляют эксклюзивный доступ к каким-либо данным в нестраничной памяти. Этот компонент доступен с уровня IRQL, меньшего или равного DISPATCH_LEVEL.

Второй компонент — объекты обратного вызова (callback). Эти объекты предоставляют возможность синхронизации кода режима ядра (на уровне IRQL меньшем или равном DISPATCH_LEVEL), а также могут синхронизировать действия двух драйверов.

Третий компонент — быстрые (fast) — мьютексы. Защищают данные на уровне APC_LEVEL и предотвращают остановку работы потока. Соответственно, доступны на уровне IRQL, меньшем или равном APC_LEVEL.

Четвертый компонент — исполнительные (executive) ресурсы. Эти ресурсы позволяют нескольким потокам читать (или же одному потоку изменять) защищенные данные. Доступны с уровня IRQL, меньшего или равного APC_LEVEL.

Пятый компонент — функции вида InterlockedXxx, выполняющие различные атомарные логические и арифметические операции над страничными данными. Доступны с любого уровня IRQL.

Шестой компонент — функции вида ExInterlockedXxx, выполняющие различные атомарные логические, арифметические операции, а также операции управления списками, являющиеся потоко-безопасными и многопроцессорно-безопасными. Все функции, кроме управления списками, доступны с любого уровня IRQL. Функции же для работы со списками (SList) доступны с уровней, меньших или равных DISPATCH_LEVEL.

Ну и, наконец, седьмое — это самые различные объекты ядра: семафоры, потоки, события и т. д., в совокупности своей используемые для осуществления различных типов синхронизации (доступных с уровня IRQL, меньшего или равного APC_LEVEL) и для синхронизации с пользовательским приложением. Ожидание доступно на уровнях IRQL, меньших или равных APC_LEVEL, сигнализирование же — меньших или равных DISPATCH_LEVEL.

Это что касается механизма синхронизации, предоставляемого операционной системой. Кроме того, драйвер и сам может определять нужные ему собственные блокировки.

Практически любому программисту, хотя бы понаслышке знающему о потоках и принципах их работы, знакома такая вещь, как "мертвые" блокировки (взаимоблокировки и т. д.). Ситуация неприятная и опасная. Специально для этого Microsoft создал утилиту Driver Verifier Deadlock Detection (DVDD), которая ищет потенциальные "мертвые" блокировки. Утилита работает очень хорошо — проверено на себе. Но доступна она, к сожалению,

только пользователям Windows XP и более ранних версий Windows; впрочем, трудно, на мой взгляд, представить драйвериста Windows 98.

И напоследок — основные рекомендации программисту, пишущему драйверы для работы на многопоточных системах:

- если весь ваш код запускается на уровне `PASSIVE_LEVEL` (`APC_LEVEL`), то используйте быстрые мьютексы, исполнительные ресурсы и объекты режима ядра — что вам удобнее;
- тестируйте драйвер тщательно и на самых различных конфигурациях аппаратного обеспечения;
- любой код, запускаемый на уровне IRQL, меньшем или равном `DISPATCH_LEVEL`, должен использовать спин-блокировки;
- используйте утилиты DVDD (упоминавшуюся выше) и CUV (Call Usage Verifier) для нахождения потенциальных "мертвых" блокировок, проверки использования спин-блокировок и множества других задач;
- используйте функции `InterlockedXxx` и `ExInterlockedXxx` для выполнения арифметических, логических и операций для работы со списками;
- определите наивысший уровень IRQL, с которого весь ваш код успешно работает, и используйте его;
- для синхронизации с пользовательским приложением используйте специально для этого созданный закрытый IOCTL.

* * *

Итак, вы ознакомились с главными советами и методиками, которые нужно применять при разработке драйверов для многопоточных систем.

Вот и все! Вы познакомились с азами написания 64-битных и многопоточных драйверов.

Глава 10

Новая операционная система Microsoft — Windows Vista

Новая операционная система от Microsoft Windows Vista — это новое поколение ОС Microsoft. По всем ее параметрам — это качественный скачок по сравнению с предыдущими версиями Windows. Улучшения коснулись абсолютно всех компонентов ОС — и интерфейса системы, и ее безопасности, и методов программирования в ней — новое API, новые способы построения Windows-приложений и сервисов. Поменялась и драйверная модель Vista — вместо WDM Microsoft предлагает WDF (Windows Driver Foundation). Но об этой модели мы подробно поговорим в следующей главе. А пока просто взглянем поближе на новую ОС и ее особенности.

ОС Vista — полностью детище инновационной платформы .NET. Что такое .NET? Это новая технология программирования от Microsoft (впрочем, на сегодняшний день не такая уж и новая), одним из главных достоинств которой является очень хорошая совместимость кода, написанного на разных языках, поддерживаемых .NET (Visual Basic .NET, C++ .NET, C#, J# — это самые популярные; создано еще огромное количество компиляторов для .NET: Lisp, PHP и т. д.). Платформа .NET состоит из двух частей: среды исполнения и средств разработки. Средств разработки — множество. Это и Visual Studio, и Eclipse, и Borland Studio... При желании (необходимости) можно писать код в обычном текстовом редакторе и компилировать его с помощью консольного компилятора. Среда исполнения — это CLR (Common Language Runtime, общеязыковая исполняющая среда). Принцип ее работы следующий.

Приложение — это, независимо от того, на каком языке оно написано, исполняемый файл (в общем случае), состоящий из двух частей: метаданных и MSIL (Microsoft Intermediate Language — так называемый "промежуточный

язык Microsoft") кода, созданного при компиляции, который CLR при исполнении уже преобразует в команды процессора (с помощью JIT-компилятора (от англ. *just-in-time* — на лету)). Как легко догадаться, этот механизм обеспечивает кросс-платформенность — достаточно только, чтобы на машине была CLR (а она сейчас создается/создана для многих аппаратных/программных платформ). CLR, кроме всего прочего, выполняет еще много функций: приведение типов, предоставление приложениям библиотеки классов .NET Framework (.NET FCL — .NET Framework Class library), обеспечение безопасности программирования и т. д.

Со стороны может показаться, что все это практически неотличимо от концепции виртуальной машины Java (JVM, Java Virtual Machine). Но есть некоторые различия. В Java используется байт-код, который создается компилятором и выполняется JVM (во время исполнения). А в .NET Framework создается машинный код на основании текста на промежуточном языке (во время исполнения). Небольшое, но достаточно существенное различие. А что коренным образом отличает .NET Framework от JVM — так это код, который преобразуется в машинный код и исполняется напрямую (а не интерпретируется во время исполнения). Вот самое главное о платформе .NET.

В ОС Vista будут входить последние версии .NET/CLR 2.0 (кодовое название — Whidbey).

Новая операционная система от Microsoft Windows Vista — преемница ОС Windows XP. Но, как оказалось позднее, ее преемственность от XP — в большой степени номинальна. Дело в том, что разработка ОС Vista, так вышло, была поделена на два этапа. С начала своей разработки Vista была основана на XP. После вынужденного перерыва в разработке Vista Microsoft вернулась к этой системе. Но заново переукомплектовывать код и переписывать его было очень трудно. Поэтому, продолжая разрабатывать Vista дальше, Microsoft взяла за основу уже Server 2003 с Service Pack 1. Таким образом, ОС Vista имеет полную совместимость с операционными системами Windows, начиная с Windows XP SP2, и основные функциональные возможности последней. Такой поворот событий дал Microsoft возможность соединить в системе все лучшее от Windows XP SP2 и Windows Server 2003 SP1.

Главные концепции новой ОС сформулированы Microsoft в 3-х словах — новом девизе новой ОС — *Clear*, *Confident*, *Connected*. Посмотрим, какие же возможности новой ОС подразумеваются под каждым из этих трех слов.

Clear (ясность, чистота) — эти слова относятся, прежде всего, к взаимодействию пользователя и системы. О чистоте и ясности говорит, в первую очередь, новый стиль пользовательского интерфейса (Aero Glass по умолчанию), выполненный в светло-голубых, прозрачных тонах. Но это также относится и к новым способам взаимодействия пользователя с системой при

работе с его персональными данными — функции простого и быстрого поиска, технологии Live Icons (живые значки), Metro и т. д. Обо всем этом подробнее далее.

Confident (конфиденциальность) — это означает качественно иной уровень в процессе обеспечения безопасной и конфиденциальной работы с компьютером. В этом направлении сделано огромное количество улучшений и нововведений: интегрированная защита от спама, spy ware и т. д., масштабное шифрование, технология UAP (User Account Protection — защита пользовательского аккаунта) и т. д. Подробнее — опять-таки далее.

Connected (соединение) — огромное количество возможностей, связанных с сетью и подсоединением устройств в ней. В Vista встроено большое количество возможностей по синхронизации информации между самыми разными типами устройств, по управлению беспроводными сетями и т. д.

Теперь рассмотрим подробнее вышеупомянутые технологии.

Live Icons (живые значки) — технология, при помощи которой значок отображает первую страницу документа. Metro — это формат документов и, одновременно, архитектура печати. Предполагается, что в Vista формат Metro будет настолько же всепроникающим, как и PDF в Mac OS X. Metro — это открытый стандарт, основанный на XML, не зависящий от устройства и приложения, включающий в себя механизм распаковки и упаковки файлов на лету (ZIP-технология). Как понятно из вышесказанного, технология Metro позволяет сохранять идеально точное форматирование документа при выводе его на печать. Metro состоит из двух компонентов: файлы на основе формата XML (название этого формата — Metro Reach) и компонента, отображающего документы Metro и управляющего ими. Microsoft предоставляет API для полноценной работы с технологией Metro.

UAP (User Account Protection) — функция, старающаяся уменьшить вред от преднамеренных (или случайных) действий пользователей, вошедших в систему под административной учетной записью (вообще, под любой в той или иной степени привилегированной учетной записью), и уменьшить степень незащищенности системы во время работы пользователя, выполняющего обычные повседневные задачи под учетной записью с высокими привилегиями. Принцип ее работы заключается в том, что при попытке выполнить любое действие, требующее высоких прав в системе, пользователю (независимо от того, под каким аккаунтом — даже если под административным) будет выведено окно, в котором будет предложено: ввести пароль администратора или, в зависимости от настроек, просто подтвердить выполнение действия. Для того чтобы в окне отображался второй вариант (подтвержде-

ние выполнения действия), необходимо сделать некоторые манипуляции в реестре системы. А именно:

1. В разделе реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System` найти ключ `ConsentPromptBehavior` с типом `DWORD` и установить его значение в 1.
2. Заново зарегистрироваться в системе.

В ранних бета-версиях Vista можно было "официально" отключить функцию UAP. Теперь же (и в более поздних версиях Vista так и будет) для ее отключения необходимо будет опять-таки сделать изменения в реестре:

1. В разделе реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System` изменить значение ключа `EnableLUA` на 0.
2. Заново зарегистрироваться в системе.

Кстати, почему ключ называется `EnableLUA`, а не `EnableUAP`, что, на первый взгляд, логичнее? Потому что функция UAP раньше называлась LUA (Limited User Account — ограниченный пользовательский аккаунт).

А теперь поговорим о трех ключевых технологиях новой ОС — Avalon, Indigo и WinFS.

Avalon — это кодовое имя технологии WPF (Windows Presentation Foundation). WPF — это новая графическая подсистема Windows. Предполагается, что она сможет полностью заменить устаревшие библиотеку user32.dll и GDI (Graphics Device Interface, интерфейс графических устройств), которые ранее полностью отвечали за графику в Windows. WDF предоставляет два вида пользовательских интерфейсов: на основе WinFX и на основе XAML (eXtensible Application Markup Language, расширенный язык разметки интерфейса Windows-приложений — языка разметки, основанного на XML). Последнее — это огромный шаг в облегчении процесса разработки приложений под Windows. Теперь создание приложений Windows сводится фактически к оперированию примитивами интерфейса и контейнерами, их содержащими. Благодаря использованию XAML, основанного на XML, и, естественно, оперирующего HTML-элементами и понятиями (тегами, например), процесс построения интерфейса Windows-приложений будет больше походить на создание кода Web-страницы.

Кроме того, теперь новая графическая подсистема работает напрямую с аппаратными средствами видеокарты, что позволяет реализовать ранее практически недостижимые быстродействие (при незначительной загрузке системных ресурсов) и разрешение изображения.

Indigo — кодовое название WCF (Windows Communication Windows). WCF — это технология для построения и запуска соединенных систем, новая архи-

тектура Web-сервисов, обеспечивающая безопасный и надежный обмен данными. Сервисно-ориентированная модель программирования в WCF упрощает разработку соединенных систем.

WinFS — это новая технология файловой системы Windows. Ее главное достоинство (кроме всех унаследованных от NTFS) — чрезвычайно эффективная система поиска. Принцип ее работы следующий. WinFS имеет базу данных, в которой содержатся описания всех файлов на диске, описания чрезвычайно точные. WinFS при получении запроса на поиск преобразует его в запрос (напоминающий SQL-запрос) к базе данных файлов на диске. Использование такого механизма позволяет составлять чрезвычайно развернутые и детализированные запросы поиска, например: "Вывести список всех фотографий формата JPEG, сделанных за период с 2002 по 2003 годы, с такими-то особенностями..." и т. д. Запрос может быть и 2 раза длиннее и еще более детализированным. Понятно, что использование такого механизма во много раз облегчает работу с документами на компьютере. Но, к сожалению, именно с этой технологией (точнее, с ее разработкой) у Microsoft возникло большое количество проблем. Для того чтобы качественно встроить WinFS в систему Windows, необходимо было сделать огромное количество изменений в архитектуре и коде системы (в частности, "перевести" систему на новое API — WinFX, о котором подробнее чуть далее). Сделать это к сроку программистам Microsoft оказалось затруднительно, поэтому решено было выпускать Vista без встроенной WinFS (по крайней мере, пока). Но этим, к сожалению, дело не закончилось. И в блогах однажды появилась надпись "WinFS is dead". Да, проекта WinFS как такового больше не существует. Все, что осталось — это некоторые его части в ASP.NET и SQL Server. Увы.

Что такое WinFX? WinFX — это новый набор API ОС Windows, отвечающий всем современным требованиям. Именно на этом API основаны Avalon и Indigo. По замыслу разработчиков, WinFX должен полностью объединить .NET и ОС. Но сложности с WinFS коснулись и WinFX, т. к. WinFS был компонентом WinFX, предназначенным для работы с данными. Тем не менее, ясно, что проект WinFX никуда не делся — иначе пришлось бы вообще сворачивать разработку ОС Vista.

Технологии Avalon, Indigo и WinFX будут доступны отдельными компонентами для всех систем Windows, начиная с Windows XP SP2 и Windows Server 2003 SP1.

Будет и серверная версия Vista — с кодовым названием Longhorn Server (пока что).

Vista будет поставляться и в виде 32-разрядной, и в виде 64-разрядной версий. В пользу применения 64-битных версий Windows есть достаточное ко-

личество аргументов — и большая надежность и безопасность (по сравнению с 32-разрядными версиями), и поддержка заметно больших объемов оперативной памяти. Но совместимость 64-битных систем пока еще оставляет желать много лучшего. Тем не менее, есть все основания надеяться, что в течение ближайшего времени эти проблемы будут в очень большой степени решены, и тогда переход на 64-разрядную платформу станет вполне оправданным, желательным и даже, быть может, необходимым шагом.

* * *

Итак, мы получили достаточно полное представление о том, что же это за операционная система, каковы ее новые возможности и т. д. В следующей главе мы приступим к главному — рассмотрим новую технологию программирования драйверов Microsoft — Windows Driver Foundation (WDF).

Глава 11

Windows Driver Foundation (WDF)

11.1. Новая драйверная модель Microsoft

До настоящего момента в этой книге мы рассматривали только старую драйверную модель Microsoft — WDM (Windows Driver Model). Но постепенно — и в этом нет никакого сомнения — актуальность этой модели будет неуклонно снижаться. Ей на смену идет WDF — новая драйверная модель Microsoft, использующая возможности новых аппаратных и программных средств, устраняющая недостатки старой модели, облегчающая процесс написания драйверов как таковой и т. д. Конечно, рассказывать о продукте, находящемся на стадии раннего бета-тестирования, — все равно, что, по меткому выражению Д. С. Платта, "стрелять по движущейся мишени". Тем не менее, я надеюсь, что мой рассказ о новой драйверной модели Microsoft (а также огромное количество необходимого и/или просто полезного материала, размещенного на прилагающемся к книге компакт-диске) поможет вам сориентироваться в этой новой технологии и уже начать свои изыскания в ней. Вы легко можете быть в курсе всех самых последних новостей о WDF, а также последних обновлений соответствующего ПО — для того достаточно подписаться на программу Microsoft о рассылке новостей по бета-тестированию этого проекта и на рассылку (скачку) бета-версий ПО. Всю необходимую информацию вы можете найти на странице **http://www.microsoft.com/whdc/driver/wdf/beta.mspx**. Итак, все предварительные пояснения сделаны, можно приступать к работе.

Прежде всего, надо сказать, что WDF-модель работает только для следующих операционных систем из семейства Windows:

- Microsoft Windows Vista;
- Microsoft Windows Server 2003;
- Microsoft Windows XP;
- Microsoft Windows 2000.

Поэтому вся изложенная здесь информация верна только для вышеперечисленных систем.

Вы уже достаточно знаете о старой модели WDM, поэтому я не буду опять расписывать ее возможности. Я только расскажу о ее недостатках, наличие которых (но, конечно, это не единственная причина, как понятно из всего вышесказанного) и привело к необходимости разработки новой драйверной модели.

Старая драйверная модель WDM страдает следующими недостатками.

- Сложность написания драйверов с использованием этой модели. Решение самых необходимых задач (вроде реализации поддержки Plug and Play) требует тысяч (!) строк кода. Очень сложны в использовании DDI (Driver Development Interface, интерфейс написания драйвера), применяющиеся для решения самых разных задач (например, синхронизации). Все эти проблемы далеко не лучшим образом сказываются на продуктивности и быстроте разработки драйверов с использованием старой модели.
- Слишком большое количество существующих разных моделей мини-портов. Это приводит к большому числу проблем: например, к необходимости знать и понимать принципы работы и устройства разных моделей мини-портов при написании драйверов для различных типов устройств.
- Большинство драйверов, написанных с применением старых технологий, могут выполняться только в режиме ядра. Проблемы, возникающие при таком подходе, очевидны: некачественно написанные драйверы могут приводить к сбоям в работе системы, злоумышленники также могут воспользоваться специально написанным драйвером для того, чтобы реализовать свои замыслы (В режиме ядра возможно все! Ну, или почти все.) и т. д.
- Обилие разных драйверных моделей приводило к трудностям при тестировании и верификации кода драйверов.

Я перечислила самые главные недостатки прежних технологий написания драйверов под Windows. А сейчас посмотрим, что хотели сделать разработчики WDF. Могу сказать, что большая часть заявленного уже сделана, остальное же, думаю, будет добавлено в самом ближайшем времени.

- Новая драйверная модель должна быть простой и гибкой. Простой — для облегчения процесса написания драйверов, гибкой — для быстрой "адаптации" к новым возможностям системы.
- Драйверная модель не должна зависеть от основных компонентов ОС (их изменение, добавление и т. д. не должно "тянуть" за собой проблемы и/или вынужденные изменения при разработке драйверов).

- Драйверная модель должна поддерживать версионность, т. е. один исполняемый файл драйвера должен работать на разных версиях ОС.
- Драйверная модель должна быть легко расширяемой.
- Драйверная модель должна позволять большинству драйверов успешно работать в пользовательском режиме.
- Драйверная модель должна поддерживать написание драйверов на языках высокого уровня.
- Драйверная модель должна позволять легко писать, анализировать, верифицировать и т. д. DDI.
- Ну и, наконец, драйверная модель должна уметь предоставлять для каждого драйвера отдельное называемое "защищенное окружение" (protected environment), или, иными словами, уметь изолировать драйвер (driver isolation).

Я, конечно, перечислила не все, а только самое главное. Но этого вполне достаточно. Теперь, я думаю, вы можете адекватно оценить истинное значение создания Microsoft новой драйверной модели.

Теперь мы можем рассмотреть WDF подробнее.

Проект WDF от Microsoft состоит из 3 "частей":

- среда для написания драйверов режима ядра (Kernel-Mode Driver Framework (KMDF));
- среда для написания драйверов пользовательского режима (User-Mode Driver Framework (UMDF));
- инструменты для проверки и отладки драйверов.

Пожалуй, самое главное в WDF — то, что эта модель в полной мере поддерживает объектно-ориентированное программирование (наборы событий, свойства и т. д. (объектная модель WDF)).

11.2. Объектная модель WDF

Каковы главные особенности объектной модели WDF?

- В этой модели объекты представляют собой "строительные блоки" для драйверов. Драйвер может изменять эти блоки через специальные интерфейсы.
- Набор событий одинаково применим ко всем типам блоков. Среда располагает стандартными обработчиками для каждого события. Если драйверу необходимо самому обработать какое-либо событие, то он создает для этой цели специальную callback-процедуру.

- Объектная модель WDF предоставляет набор объектов, содержащий объекты (прошу прощения за тавтологию), общие для любых драйверов — такие как устройства, очереди и т. д.
- У каждого объекта есть свойства (properties), которые определяют характеристики объекта. Для каждого свойства определен свой метод, работающий с ним.
- Все объекты организованы в иерархическую систему. В ее вершине стоит главный WDF-объект устройства, также являющийся родительским объектом по умолчанию для всех объектов, для которых таковой не указан. Практически для любого типа создаваемого дочернего объекта драйвер может указать нужный объект-"родитель". Но некоторые типы объектов обладают неизменяемыми дефолтными объектами-"родителями". Удаление родительского объекта автоматически приводит к удалению всех дочерних объектов.

Объектная модель относится как к KMDF, так и UMDF. Но WDF-объекты в этих двух средах осуществлены по-разному. Рассмотрим оба варианта.

11.3. Объекты KMDF

Объекты режима ядра непрозрачны для драйвера, и он никогда не имеет прямого к ним доступа. Драйвер может выполнять какие-либо действия с объектом только с помощью указателя.

Далее приведена табл. 11.1 из MSDN, в которой перечисляются наиболее употребительные WDF-объекты режима ядра (перевод таблицы авторский).

Таблица 11.1. *WDF-объекты режима ядра*

Имя типа объекта	Назначение
`WDFDRIVER`	Представляет объект драйвера
`WDFDEVICE`	Представляет объект устройства
`WDFQUEUE`	Представляет очередь I/O-запросов (ввода/вывода)
`WDFINTERRUPT`	Представляет ресурсы прерывания
`WDFREQUEST`	Представляет I/O-запрос (ввода/вывода)
`WDFMEMORY`	Представляет буфер для I/O-запроса (ввода/вывода)
`WDFDMAENABLER`	Описывает характеристик всех DMA-передач устройства

Таблица 11.1 *(окончание)*

Имя типа объекта	Назначение
WDFDMATRANSACTION	Управляет операциями для отдельного DMA-запроса
WDFIOTARGET	Представляет целевой драйвер I/O-запроса (ввода/вывода)

Все объекты режима ядра уникальны. Ими невозможно управлять с помощью функций семейства `Ob`*`Xxx`*. Также ими невозможно управлять с помощью менеджера объектов Windows. Создание и управление этими объектами возможно только для самой среды и для WDF-драйверов.

11.4. Объекты UMDF

Сущность UMDF-объектов — это COM. UMDF-объекты используют подмножество COM для реализации интерфейсов запросов и прочего. В драйверах пользовательского режима как драйвер, так и среда реализуют и предоставляют интерфейсы в стиле COM. Само собой, отпадает необходимость использования указателей.

Разных типов объектов UMDF меньше, чем KMDF. Это понятно — в пользовательском режиме многие действия (а значит, и объекты для работы с ними) запрещены. Опять-таки, таблица (те же замечания, что и к предыдущей) — табл. 11.2.

Таблица 11.2. *Объекты UMDF*

Имя объекта интерфейса	Назначение
IWDFObject	Определяет базовый тип WDF-объекта
IWDFDriver	Представляет объект драйвера
IWDFDevice	Представляет объект устройства
IWDFFile	Представляет объект файла
IWDFIoQueue	Представляет очередь I/O-запросов (ввода/вывода)
IWDFIoRequest	Описывает I/O-запрос (ввода/вывода)
IWDFIoTarget	Представляет целевой драйвер I/O-запроса (ввода/вывода)
IWDFMemory	Предоставляет доступ к области памяти

Теперь поговорим о поддержке технологии Plug and Play и технологии управления питанием, а также о модели ввода/вывода в WDF.

11.5. Plug and Play, управление питанием и модель ввода/вывода в WDF

Как я уже говорила, поддержка технологий управления питанием и Plug and Play в предыдущей драйверной модели осуществлялась очень непросто, приходилось писать огромное количество строк кода, решать множество проблем и т. д. WDF предлагает следующие концепции для решения этих проблем:

- драйвер вовсе не обязан обрабатывать и/или отвечать на все входящие запросы. Ему должна быть предоставлена возможность отвечать только на те запросы, которые ему нужны;
- среда должна предоставлять стандартные обработчики для огромного количества возможностей Plug and Play: остановка и запуск устройства, удаление его и т. д.;
- действия WDF в любой момент времени должны быть четко определенными и предсказуемыми;
- обе технологии должны быть полностью интегрированы с остальными частями среды;
- среда должна поддерживать как простой, так и сложный дизайн устройств и драйверов;
- драйвер должен иметь возможность переопределить любые установленные средой значения (чего бы то ни было) по умолчанию.

WDF интегрирует технологии Plug and Play и управления питанием с технологией организации очередей I/O-запросов, а последнюю, в свою очередь, с технологией отмены запросов.

Еще одна новая интересная вещь — управление параллельными операциями. Для того чтобы устранить проблемы с одновременным доступом к объектам, возможной порче данных при этом и т. д., WDF предоставляет несколько внутренних механизмов синхронизации, а также создает и удерживает все необходимые для драйвера в данный момент блокировки.

Теперь об I/O-модели (модели ввода/вывода).

В Windows IRP — это главный механизм связи между операционной системой и драйверами (а также между самими драйверами), основанный на пакетном механизме. I/O-менеджер (менеджер ввода/вывода) посылает IRP

драйверам для уведомления драйверов о запросах Plug and Play, изменениях в состоянии устройства и т. д. Но модель ввода/вывода в WDF — это нечто большее, чем просто механизм передачи данных.

Среда управляет механизмами управления, завершения, организации очереди IRP для всех WDF-драйверов. Среда вызывает callback-процедуры драйвера для того, чтобы уведомить его о важных событиях (таких, например, как наличие запросов, которые драйвер должен обработать).

После получения запроса среда записывает о нем всю необходимую информацию, если необходимо, создает WDF-объект для представления запроса, а затем вызывает нужную callback-процедуру драйвера для дальнейшей обработки запроса.

Такие объекты WDF, как очереди (queues), помогают драйверам управлять потоком запросов. При необходимости драйвер может создать нужное ему количество очередей и отконфигурировать их по своему усмотрению (например, для того чтобы запросы определенного типа поступали в отведенные им очереди). Также драйвер может назначить для каждой очереди механизм обработки. В зависимости от этого структура будет или немедленно доставлять определенные запросы драйверу, или же помещать их в очередь для дальнейшей обработки. Среда отслеживает каждый запрос, "владельцем" которого является драйвер, до тех пор, пока он не будет отменен, обработан или переслан другому обработчику. Ввиду того, что среда осведомлена обо всех активных запросах, при необходимости (например, в случае удаления устройства) она может вызвать соответствующую callback-функцию драйвера.

Заканчивая рассказ о модели ввода/вывода, приведу диаграмму потока данных в этой модели (из MSDN, с авторским переводом). И KMDF, и UMDF используют одинаковую модель ввода/вывода (хотя она и реализуется разными компонентами) (рис. 11.1).

С I/O моделью, думаю, все ясно.

Что, вообще говоря, делают и содержат эти frameworks — и KMDF, и UMDF? Главных их особенностей несколько:

- управление жизненным циклом объектов;
- управление потоком запросов ввода/вывода и уведомлениями питания и Plug and Play;
- определение объектов WDF, которые могут быть инстанцированы WDF-драйверами;
- предоставление набора DDI-функций, которые могут использоваться WDF-драйверами для управления объектами.

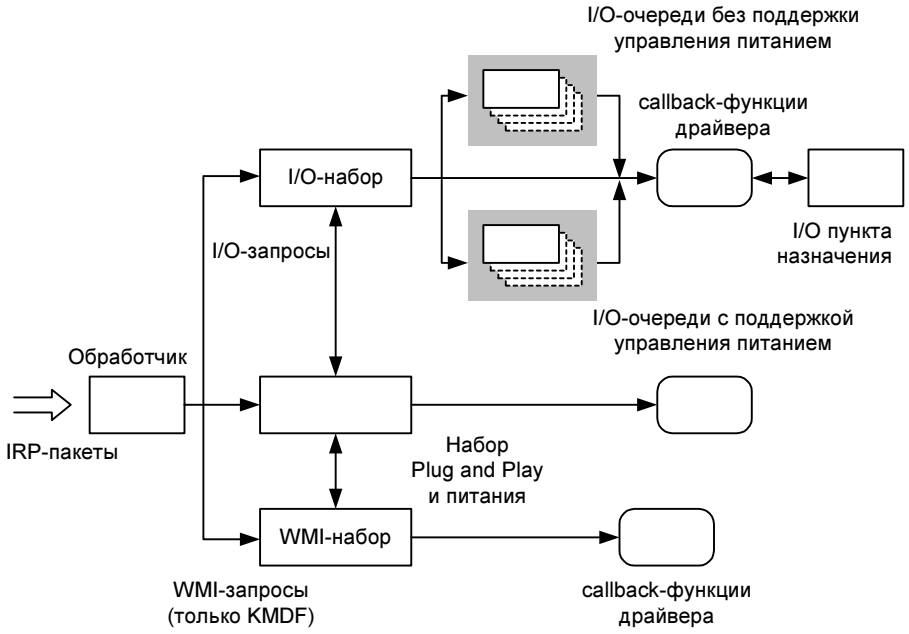

Рис. 11.1. Модель ввода/вывода UMDF (KMDF)

А теперь перейдем к особенностям отдельно KMDF и UMDF.

Подробно говорить о каждой среде мы будем позднее, а здесь отметим только, что KMDF поддерживает следующие типы kernel-драйверов:

- ❒ шинные драйверы для стека устройств Plug and Play;
- ❒ фильтр-драйверы для устройств Plug and Play;
- ❒ legacy-драйверы для устройств, не включенных в стек Plug and Play;
- ❒ функциональные драйверы для устройств Plug and Play.

Ну, а с UMDF все ясно. Поддерживаемые типы драйверов — пользовательские.

* * *

А теперь приступим собственно к написанию драйверов.

Глава 12

Написание драйверов в Vista — KMDF

В этой главе я расскажу о написании драйверов режима ядра для новой ОС Vista с использованием среды KMDF.

Для каких типов устройств (с теми или иными замечаниями в отдельных случаях) мы можем писать драйверы, используя KMDF?

- Legacy-устройства (не Plug and Play).
- USB, ISA, PCI, PCMCIA, SD, soft-modem, IEEE 1394.
- NDIS WDM и протокол-драйверы.
- Драйверы устройств хранения данных (storage).
- Winsock и TDI (transport drover interface) драйверы.

Из каких компонентов состоит KMDF?

- Библиотеки.
- Инструменты.
- Примеры драйверов.
- Заголовочные файлы.
- Pdb-файлы с отладочными символами (symbols).
- Файлы формата трассировки.

Пробежались по "внешнему" облику KMDF. Теперь поговорим о такой вещи, как объектная модель KMDF.

12.1. Объектная модель KMDF

KMDF определяет объектно-ориентированную среду программирования драйверов. Каждый объект экспортирует методы и свойства, которые могут быть использованы драйвером.

Мы разберем ключевые понятия объектно-ориентированной среды KMDF:

- *методы* — это функции, выполняющие какие-либо действия над объектом;
- *свойства* — это функции, читающие и записывающие данные из/в поля объекта;
- *события* позволяют получать информацию обо всех происходящих ситуациях, с которыми они ассоциированы.

Все объекты KMDF организованы в иерархию. Типов объектов в KMDF очень много: это и строки, и прерывания, и коллекции, и строки реестра, и т. д. У каждого объекта есть 5 атрибутов, описывающих его размер, родительский объект, уровень исполнения и проч.

Каков "жизненный цикл" объекта KMDF?

1. KMDF выделяет память для объекта и его контекста из пула нестраничной памяти.
2. KMDF инициализирует атрибуты объекта нулевыми значениями и/или в соответствии со спецификациями драйвера.
3. KMDF обнуляет контекст объекта.
4. KMDF конфигурирует объект: устанавливает характеристики, специфичные для этого объекта и т. д.

А теперь — к простейшему KMDF-драйверу.

12.2. Простейший KMDF-драйвер

Рассмотрим "скелет" простейшего KMDF-драйвера (папка src стандартной установки KMDF).

Итак, каковы главные компоненты драйвера? Перечислю их:

- функция `DriverEntry` — главная точка входа драйвера;
- callback-событие `EvtDriverDeviceAdd`, создающее объект устройства, его интерфейс;
- callback-функции ввода/вывода для чтения, записи, и управления устройством.

Рассмотрим, как они выглядят в коде. И начнем с функции `DriverEntry` (листинг 12.1).

Листинг 12.1. Функция `DriverEntry`

```
NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT  DriverObject,
  IN PUNICODE_STRING  RegistryPath
);
```

Здесь все ясно — такую функцию `DriverEntry` мы уже разбирали. Посмотрим, как эта функция выглядит и что делает в коде простейшего драйвера, входящего в стандартную поставку KMDF (листинг 12.2).

Листинг 12.2. Функция `DriverEntry` sample-драйвера

```
NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT  DriverObject,
  IN PUNICODE_STRING RegistryPath
)

{
  NTSTATUS status = STATUS_SUCCESS; // по умолчанию статус — успех
  WDF_DRIVER_CONFIG   config;   // объект конфигурирования драйвера

  KdPrint(("Toaster Function Driver Sample — ",
           "Driver Framework Edition.\n")); // отладочное сообщение
  KdPrint(("Built %s %s\n", __DATE__, __TIME__)); // выводим версию,
                                    // формируемую из даты и времени

  WDF_DRIVER_CONFIG_INIT( // инициализация конфигурации
    &config,
    ToasterEvtDeviceAdd
  );
```

```
  status = WdfDriverCreate(   // создаем WDF-объект драйвера
    DriverObject,             // объект драйвера
    RegistryPath,             // путь в реестре
    WDF_NO_OBJECT_ATTRIBUTES, // атрибуты драйверов
    &config,        // информация конфигурирования драйвера
    WDF_NO_HANDLE
  );

  if (!NT_SUCCESS(status)) {   // если не удалось
    KdPrint(("WdfDriverCreate failed with "
             "status 0x%x\n", status)); // выводим отладочное
                                        // сообщение со статусом
  }
  return status; // возвращаем статус
}
```

Все предельно ясно и понятно — инициализируем, создаем объект драйвера и т. д.

А теперь — callback-событие `EvtDriverDeviceAdd` (листинг 12.3).

Листинг 12.3. Callback-событие `EvtDriverDeviceAdd`

```
NTSTATUS
ToasterEvtDeviceAdd(
  IN WDFDRIVER        Driver,
  IN PWDFDEVICE_INIT  DeviceInit
)
{
  NTSTATUS                status = STATUS_SUCCESS;  // переменная статуса
  PFDO_DATA               fdoData;           // данные FDO
  WDF_IO_QUEUE_CONFIG     queueConfig;       // конфигурирование очереди
  WDF_OBJECT_ATTRIBUTES   fdoAttributes;     // атрибуты FDO
  WDFDEVICE               hDevice;           // WDF-устройство
  WDFQUEUE                queue;             // очередь

  UNREFERENCED_PARAMETER(Driver);
```

```
PAGED_CODE();                // код организуется в страницу памяти
KdPrint(("ToasterEvtDeviceAdd called\n")); // отладочное сообщение

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&fdoAttributes,
  FDO_DATA);     // инициализация атрибутов
                 // и контекста объекта устройства

status = WdfDeviceCreate(&DeviceInit, &fdoAttributes,
  &hDevice);                 // создаем WDF-объект устройства
if (!NT_SUCCESS(status)) {   // проверяем статус
  KdPrint(("WdfDeviceCreate failed with status code",
           " 0x%x\n", status)); // выводим сообщение со статусом
  return status;               // возвращаем его
}

fdoData = ToasterFdoGetData(hDevice); // получаем контекст устройства
status = WdfDeviceCreateDeviceInterface( // создаем интерфейс
                                         // устройства
           hDevice,
           (LPGUID) &GUID_DEVINTERFACE_TOASTER,
           NULL
         );

if (!NT_SUCCESS (status)) {    // проверяем статус
  KdPrint(("WdfDeviceCreateDeviceInterface failed",
           "0x%x\n", status)); // выводим сообщение со статусом
  return status;               // возвращаем его
}

WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&queueConfig,
  WdfIoQueueDispatchParallel);  // конфигурируем очередь
                                // по умолчанию

queueConfig.EvtIoRead = ToasterEvtIoRead;
queueConfig.EvtIoWrite = ToasterEvtIoWrite;
```

```
  queueConfig.EvtIoDeviceControl =
    ToasterEvtIoDeviceControl;
  // в приведенных выше строках мы регистрируем функции
  // обратного вызова для обработки запросов ввода/вывода
  status = WdfIoQueueCreate(           // создаем очередь
              hDevice,
              &queueConfig,
              WDF_NO_OBJECT_ATTRIBUTES,
              &queue
  );

  if (!NT_SUCCESS (status)) {   // проверяем статус
    // отладочное сообщение
    KdPrint( ("WdfIoQueueCreate failed 0x%x\n", status));
    return status;              // возвращаем статус
  }

  return status;                // возвращаем статус
}
```

Выше (см. листинг 12.3) мы использовали структуру `FDO_DATA`. Рассмотрим ее определение (листинг 12.4).

Листинг 12.4. Структура `FDO_DATA`

```
typedef struct _FDO_DATA
{
  WDFWMIINSTANCE  WmiDeviceArrivalEvent;
  BOOLEAN         WmiPowerDeviceEnableRegistered;
  TOASTER_INTERFACE_STANDARD  BusInterface;
} FDO_DATA, *PFDO_DATA;

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME (FDO_DATA, ToasterFdoGetData)
```

У всех параметров — интуитивные названия. А вот `WDF_DECLARE_CONTEXT_TYPE_WITH_NAME` — это макрос, инициализирующий

атрибуты FDO и устанавливающий указатели на имя, длину контекста объекта и на сам контекст объекта.

Все, с этим разобрались. Перейдем к последнему — обработке I/O-запросов.

Драйвер может иметь следующие функции для обработки поступающих I/O-запросов:

- `EvtIoRead`;
- `EvtIoWrite`;
- `EvtIoDeviceControl`;
- `EvtIoInternalDeviceControl`;
- `EvtIoDefault`.

Здесь все видно — чтение, запись, обработчик по умолчанию и т. д. Все эти запросы "скапливаются" в соответствующие очереди, для каждой из которых драйвер может зарегистрировать одну/несколько callback-функций.

Итак, мы рассмотрели скелет простейшего KMDF-драйвера и теперь можем приступить к "разбору" структуры и общего вида нормального KMDF-драйвера устройства.

Первое, что мы должны сделать — реализовать обработку прерываний. Для этого нужно создать объект прерывания, включить/отключить его и выполнить все (возможно) нужные дополнительные инициализации. После этого осталось только, собственно говоря, обработать прерывания.

Объект прерывания создается в функции обратного вызова `EvtDriverDeviceAdd`. Понятно, что драйвер должен иметь созданный объект прерывания абсолютно для *каждого* объекта прерывания, который он собирается обрабатывать.

Вот стандартный примерный код, создающий прерывание (листинг 12.5).

Листинг 12.5. Создаем объект прерывания

```
WDF_INTERRUPT_CONFIG_INIT(&interruptConfig,
                          NICEvtInterruptIsr,
                          NICEvtInterruptDpc);

interruptConfig.EvtInterruptEnable = NICEvtInterruptEnable;
interruptConfig.EvtInterruptDisable = NICEvtInterruptDisable;

status = WdfInterruptCreate(FdoData->WdfDevice,
// самый главный вызов — создаем прерывание
```

```
                              &interruptConfig,
                              WDF_NO_OBJECT_ATTRIBUTES,
// не указываем атрибутов
                              &FdoData->WdfInterrupt);

if (!NT_SUCCESS (status)) // проверяем статус проделанной операции
{
  return status;
}
```

Как известно (и как уже отмечалось ранее), объекты прерывания (читай — и сами прерывания, ассоциированные с этим объектом) могут быть включены и отключены (листинги 12.6 и 12.7).

Листинг 12.6. Включение объекта прерывания

```
NTSTATUS
NICEvtInterruptEnable(
  IN WDFINTERRUPT  Interrupt,       // объект прерывания
  IN WDFDEVICE     AssociatedDevice // ассоциированное устройство
)
{
  PFDO_DATA        fdoData;

  fdoData = FdoGetData(AssociatedDevice);
  NICEnableInterrupt(Interrupt, fdoData);

  return STATUS_SUCCESS;   // возвращаем статус успешного выполнения
}
```

Листинг 12.7. Отключение объекта прерывания

```
NTSTATUS
NICEvtInterruptDisable(
  IN WDFINTERRUPT  Interrupt,       // объект прерывания
  IN WDFDEVICE     AssociatedDevice // ассоциированное устройство
)
```

```
{
  PFDO_DATA        fdoData;

  fdoData = FdoGetData(AssociatedDevice);
  NICDisableInterrupt(fdoData);

  return STATUS_SUCCESS; // возвращаем статус успешного выполнения
}
```

Сама обработка прерывания происходит с помощью двух функций обратного вызова — `EvtInterruptISR` и `EvtInterruptDPC`, для обработки ISR и DPC соответственно. Эти функции представлены в листингах 12.8 и 12.9.

Листинг 12.8. Функция `EvtInterruptISR`

```
BOOLEAN
NICEvtInterruptIsr(
  IN WDFINTERRUPT Interrupt, // объект прерывания
  IN ULONG        MessageID
)
{
  BOOLEAN    InterruptRecognized = FALSE;
  PFDO_DATA  FdoData = NULL;
  USHORT     IntStatus;

  UNREFERENCED_PARAMETER(MessageID);

  FdoData = FdoGetData(WdfInterruptGetDevice(Interrupt));

  if (!NIC_INTERRUPT_DISABLED(FdoData) &&
       NIC_INTERRUPT_ACTIVE(FdoData)) // проверяем, является ли
                                      // прерывание активным и включенным
  {
    InterruptRecognized = TRUE;

    NICDisableInterrupt(FdoData); // выключаем прерывание
```

```
    NIC_ACK_INTERRUPT(FdoData, IntStatus);
    WdfInterruptQueueDpcForIsr(Interrupt);
    // определяем, что это за прерывание, и получаем его статус
  }
  return InterruptRecognized;  // возвращаем опознанное прерывание
}
```

Листинг 12.9. Функция `EvtInterrupdDpc`

```
VOID
NICEvtInterruptDpc(
  IN WDFINTERRUPT WdfInterrupt, // объект прерывания
  IN WDFOBJECT    WdfDevice     // объект устройства
)

{
  PFDO_DATA fdoData = NULL;

  fdoData = FdoGetData(WdfDevice);

  WdfSpinLockAcquire (fdoData->RcvLock);
  NICHandleRecvInterrupt(fdoData);
  WdfSpinLockRelease (fdoData->RcvLock);

  WdfSpinLockAcquire (fdoData->SendLock);
  NICHandleSendInterrupt(fdoData);
  WdfSpinLockRelease (fdoData->SendLock);

  NICCheckForQueuedSends(fdoData);

  WdfSpinLockAcquire (fdoData->RcvLock);
  NICStartRecv(fdoData);
  WdfSpinLockRelease (fdoData->RcvLock);
  WdfInterruptSynchronize(
        WdfInterrupt,
```

```
        NICEnableInterrupt,
        fdoData);

  return;
}
```

В приведенном выше коде все предельно ясно — установка блокировок, включение объекта прерывания и т. д. Дополнительных пояснений, я думаю, не требуется.

Как драйвер "забирает" для своего устройства все необходимое, а потом возвращает это системе? С помощью опять-таки двух функций все того же обратного вызова: `EvtDevicePrepareHardware` и `EvtDeviceReleaseHardware` для "захвата" и освобождения соответственно.

Код из листинга 12.10 выполняет "захват" нужных устройству ресурсов.

Листинг 12.10. Функция `MapHwResources`

```
NTSTATUS
NICMapHWResources(
  IN OUT PFDO_DATA FdoData,
  WDFCMRESLIST   ResourcesTranslated
)
{
  PCM_PARTIAL_RESOURCE_DESCRIPTOR descriptor;
  ULONG       i;
  NTSTATUS    status = STATUS_SUCCESS;
  BOOLEAN     bResPort      = FALSE;
  BOOLEAN     bResInterrupt = FALSE;
  BOOLEAN     bResMemory    = FALSE;
  ULONG       numberOfBARs  = 0;

  PAGED_CODE();

  for (i=0;   // прокручиваем цикл до конца списка ресурсов
       i<WdfCmResourceListGetCount(ResourcesTranslated);
       i++) {
```

```
descriptor =
  WdfCmResourceListGetDescriptor(ResourcesTranslated, i);

if(!descriptor){   // если ничего
  return STATUS_DEVICE_CONFIGURATION_ERROR; // то ошибка!
}

switch (descriptor->Type) { // если же пока все нормально и мы
  // получили указатель на ресурс, который нужно захватить

case CmResourceTypePort: // то начинаем перебирать варианты;
                         // если это порт
  numberOfBARs++;        // увеличиваем счетчик

  if (numberOfBARs != 2) { // должно быть два!
    status = STATUS_DEVICE_CONFIGURATION_ERROR; // ошибка ...
    return status;         // возвращаем статус ...
  }

  FdoData->IoBaseAddress =   // базовый адрес ввода/вывода
    ULongToPtr(descriptor->u.Port.Start.LowPart);
  FdoData->IoRange = descriptor->u.Port.Length; // диапазон
  FdoData->ReadPort = NICReadPortUShort;   // функция чтения порта
  FdoData->WritePort = NICWritePortUShort; // функция записи порта

  bResPort = TRUE;
  FdoData->MappedPorts = FALSE;
  break;   // выходим

case CmResourceTypeMemory:  // если же мы выделяем память
  numberOfBARs++;           // увеличиваем счетчик
  if (numberOfBARs == 1) {  // если один
    ASSERT(descriptor->u.Memory.Length == 0x1000); // пространство
                                                   // памяти 0x1000
    FdoData->MemPhysAddress =
             descriptor->u.Memory.Start;  // старт "отрезка"
```

```
        FdoData->CSRAddress = MmMapIoSpace( // регистрация участка памяти
                                descriptor->u.Memory.Start,
                                NIC_MAP_IOSPACE_LENGTH,
                                MmNonCached);
        bResMemory = TRUE;
      } else if(numberOfBARs == 2){ // если два
        FdoData->IoBaseAddress = MmMapIoSpace(
                          descriptor->u.Memory.Start,
                          descriptor->u.Memory.Length,
                          MmNonCached);  // регистровая память
        FdoData->ReadPort = NICReadRegisterUShort;
        FdoData->WritePort = NICWriteRegisterUShort;
        FdoData->MappedPorts = TRUE;
        bResPort = TRUE;

      } else if(numberOfBARs == 3){   // flash-память
          //ASSERT(descriptor->u.Memory.Length ==
          //                     0x100000);
      } else {
          status = STATUS_DEVICE_CONFIGURATION_ERROR;
          return status;
      }

      break;

    case CmResourceTypeInterrupt: // если прерывание
      ASSERT(!bResInterrupt);
      bResInterrupt = TRUE;
      break;

    default:   // стандартное действие
      break;   // выходим
    }
  }
```

```
  // проверяем, все ли успешно "захватили", что надо
  if (!(bResPort && bResInterrupt && bResMemory)) {
    status = STATUS_DEVICE_CONFIGURATION_ERROR; // если нет — ошибка
  }
  ...
  return status; // возвращаем статус
}
```

Так... Нам осталось разобраться с очередями и работой с реестром — это главные нюансы, которые мы еще не рассмотрели.

Что такое очереди — уже не раз рассматривалось на страницах этой книги. Итак, сейчас мы рассмотрим разнообразные очереди, в которые "сваливаются" разнообразные же запросы ввода/вывода.

Очереди бывают последовательные, параллельные и так называемые "ручные" (manual).

Приступим к созданию и конфигурированию параллельной очереди.

Код из листинга 12.11 создает параллельную очередь для накапливания запросов записи.

Листинг 12.11. Создаем параллельную очередь "записи"

```
NTSTATUS                status;        // статус
WDF_IO_QUEUE_CONFIG     ioQueueConfig; // конфигурирование очереди
WDF_OBJECT_ATTRIBUTES   attributes;    // атрибуты

...

WDF_IO_QUEUE_CONFIG_INIT( // инициализация конфигурирования
                          // очереди...
  &ioQueueConfig,
  WdfIoQueueDispatchParallel // будем делать параллельную!
);

ioQueueConfig.EvtIoWrite = PciDrvEvtIoWrite;

status = WdfIoQueueCreate(       // создаем собственно очередь
           FdoData->WdfDevice,
```

```
            &ioQueueConfig,
            WDF_NO_OBJECT_ATTRIBUTES,
            &FdoData->WriteQueue  // указатель на очередь
);

if (!NT_SUCCESS (status)) {   // проверяем статус
  return status;              // и возвращаем его
}

// конфигурируем обработку запросов
status = WdfDeviceConfigureRequestDispatching(
  FdoData->WdfDevice,
  FdoData->WriteQueue,
  WdfRequestTypeWrite);

if(!NT_SUCCESS (status)){   // проверяем статус
  ASSERT(NT_SUCCESS(status));
  return status;            // и как всегда возвращаем
}
```

Этим кодом мы создали параллельную очередь. А вот так можно создать "ручную" (листинг 12.12).

Листинг 12.12. Создаем "ручную" очередь

```
WDF_IO_QUEUE_CONFIG_INIT(  // инициализация конфигурирования очереди ...
  &ioQueueConfig,
  WdfIoQueueDispatchManual // будем делать "ручную"!
);

status = WdfIoQueueCreate (
            FdoData->WdfDevice,
            &ioQueueConfig,
            WDF_NO_OBJECT_ATTRIBUTES,
            &FdoData->PendingWriteQueue
);
```

```
if(!NT_SUCCESS (status)){
  return status;
}
```

Общий принцип, думаю, ясен. Посмотрим, как обрабатывать запросы из очереди (листинг 12.13).

Листинг 12.13. Обработка запросов из очереди

```
VOID
PciDrvEvtIoDeviceControl(
  IN WDFQUEUE     Queue,              // очередь
  IN WDFREQUEST   Request,            // запрос
  IN size_t       OutputBufferLength, // длина выходного буфера
  IN size_t       InputBufferLength,  // длина входящего буфера
  IN ULONG        IoControlCode
)
{
  NTSTATUS                status= STATUS_SUCCESS;
  PFDO_DATA               fdoData = NULL;
  WDFDEVICE               hDevice;
  WDF_REQUEST_PARAMETERS  params;

  UNREFERENCED_PARAMETER(OutputBufferLength);
  UNREFERENCED_PARAMETER(InputBufferLength);

  hDevice = WdfIoQueueGetDevice(Queue); // получаем устройство
  fdoData = FdoGetData(hDevice);        // структура FDO_DATA

  WDF_REQUEST_PARAMETERS_INIT(&params);

  WdfRequestGetParameters( // получаем параметры запроса
    Request,
    &params
  );
```

```
switch (IoControlCode)   // переключатель
{
  case IOCTL_NDISPROT_QUERY_OID_VALUE: // проверяем варианты
    ASSERT((IoControlCode & 0x3) == METHOD_BUFFERED);
    NICHandleQueryOidRequest(
                      Queue,
                      Request,
                      &params
    );
    break;

    ...
  case IOCTL_NDISPROT_INDICATE_STATUS: // проверяем варианты
    status = WdfRequestForwardToIoQueue(Request,
               // перенаправляем в очередь
               fdoData->PendingIoctlQueue);
    if(!NT_SUCCESS(status)){   // проверяем статус
      WdfRequestComplete(Request, status); // обработка завершена
     break; // выход
    }
    break;  // выход

  default:  // стандартный вариант
    // говорим, что неизвестный IOCTL
    ASSERTMSG(FALSE, "Invalid IOCTL request\n");
    WdfRequestComplete(Request,  // обработка завершена
                       STATUS_INVALID_DEVICE_REQUEST);
    break;
  }
return;
```

Этот код "забирает" запросы из параллельной очереди. А вот код, делающий то же самое с "ручной" очередью (листинг 12.14).

Листинг 12.14. Выбираем запросы из "ручной" очереди

```
NICGetIoctlRequest(        // получаем запрос IOCTL
  IN WDFQUEUE Queue,       // очередь
  IN ULONG FunctionCode,
  OUT WDFREQUEST*  Request // сам запрос
)
{
  NTSTATUS            status = STATUS_UNSUCCESSFUL; // статус
  WDF_REQUEST_PARAMETERS params;                    // параметр
  WDFREQUEST          tagRequest;
  WDFREQUEST          prevTagRequest;

  WDF_REQUEST_PARAMETERS_INIT(&params);

  *Request = NULL;
  prevTagRequest = tagRequest = NULL;

  do { // начинаем ...
    WDF_REQUEST_PARAMETERS_INIT(&params);
    status = WdfIoQueueFindRequest(Queue, // поиск запроса в очереди
                                   prevTagRequest,
                                   NULL,
                                   &params,
                                   &tagRequest);

    if(prevTagRequest) {
      WdfObjectDereference(prevTagRequest);
    }
    if(status == STATUS_NO_MORE_ENTRIES) { // разные варианты
                                           // результата
      status = STATUS_UNSUCCESSFUL;
      break;
    }
```

```
if(status == STATUS_NOT_FOUND) { // не найден
  prevTagRequest = tagRequest = NULL;
  continue;
}

if(!NT_SUCCESS(status)) { // статус
  status = STATUS_UNSUCCESSFUL;
  break;
}

if(FunctionCode ==        // функциональный код
   params.Parameters.DeviceIoControl.IoControlCode){
   status = WdfIoQueueRetrieveFoundRequest(
              Queue,
              tagRequest,
              Request
            );

   WdfObjectDereference(tagRequest);

   if(status == STATUS_NOT_FOUND) { // не найден
     prevTagRequest = tagRequest = NULL;
     continue;
   }

   if(!NT_SUCCESS(status)) { // статус
     status = STATUS_UNSUCCESSFUL;
     break;
   }

   ASSERT(*Request == tagRequest);
   status = STATUS_SUCCESS;
   break;

}
```

```
    else {
      prevTagRequest = tagRequest;
      continue;
    }

  } while (TRUE);   // выполняем, пока истинно
  return status;
}
```

Ну, и последний код в теме очередей — это поиск в "ручной" очереди (листинг 12.15).

Листинг 12.15. Поиск в "ручной" очереди

```
NICGetIoctlRequest(        // получаем запрос
  IN WDFQUEUE      Queue,          // очередь
  IN ULONG         FunctionCode,   // функциональный код
  OUT WDFREQUEST*  Request         // запрос
)
{
  NTSTATUS            status = STATUS_UNSUCCESSFUL;
  WDF_REQUEST_PARAMETERS params;
  WDFREQUEST          tagRequest;
  WDFREQUEST          prevTagRequest;

  WDF_REQUEST_PARAMETERS_INIT(&params);

  *Request = NULL;
  prevTagRequest = tagRequest = NULL;

  do {
    WDF_REQUEST_PARAMETERS_INIT(&params);
    status = WdfIoQueueFindRequest(Queue, // поиск запроса
                    prevTagRequest,
                    NULL,
                    &params,
                    &tagRequest);
```

```
if(prevTagRequest) {
  WdfObjectDereference(prevTagRequest);
}
if(status == STATUS_NO_MORE_ENTRIES) {
  status = STATUS_UNSUCCESSFUL;
  break;
}

if(status == STATUS_NOT_FOUND) {
  prevTagRequest = tagRequest = NULL;
  continue;
}

if( !NT_SUCCESS(status)) {
  status = STATUS_UNSUCCESSFUL;
  break;
}

if(FunctionCode ==   // функциональный код
  params.Parameters.DeviceIoControl.IoControlCode){
  status = WdfIoQueueRetrieveFoundRequest(
             Queue,
             tagRequest,   // TagRequest
             Request
           );

  WdfObjectDereference(tagRequest);

  if(status == STATUS_NOT_FOUND) {
    prevTagRequest = tagRequest = NULL;
    continue;
  }

  if( !NT_SUCCESS(status)) {
    status = STATUS_UNSUCCESSFUL;
```

```
        break;
      }

      ASSERT(*Request == tagRequest);
      status =  STATUS_SUCCESS;
      break;

    }else {
      prevTagRequest = tagRequest;
      continue;
    }

  } WHILE (TRUE);
  return status;
}
```

Перейдем к реестру (а точнее, к работе с ним). Код из листинга 12.16 читает реестр.

Листинг 12.16. Чтение записи в реестре

```
BOOLEAN
PciDrvReadFdoRegistryKeyValue(   // читаем ключ в реестре (FDO)
  __in  PWDFDEVICE_INIT  DeviceInit,
  __in  PWCHAR           Name,   // имя
  __out PULONG           Value   // значение
)
{
  WDFKEY      hKey = NULL;
  NTSTATUS    status;
  BOOLEAN     retValue = FALSE;
  UNICODE_STRING  valueName;

  PAGED_CODE(); // странично организованный код

  *Value = 0;
```

```
  status = WdfFdoInitOpenRegistryKey(DeviceInit, // открываем ключ реестра
                PLUGPLAY_REGKEY_DEVICE,
                STANDARD_RIGHTS_ALL,
                WDF_NO_OBJECT_ATTRIBUTES,
                &hKey);

  if (NT_SUCCESS (status)) { // если успех
    RtlInitUnicodeString (&valueName,Name);

    status = WdfRegistryQueryULong (hKey, &valueName, Value);

    if (NT_SUCCESS (status)) {
      RetValue = TRUE;
    }

    WdfRegistryClose(hKey); // закрываем ключ
  }
  return retValue;          // и возвращаем значение
}
```

* * *

Итак, ну все, хватит. Всей вышеприведенной информации должно "с головой" хватить для того, чтобы легко разобраться со всеми дополнительными исходными кодами, приведенными (для вашего удобства) в конце книги *(см. приложения 2—4)*. Едем дальше!

Глава 13

Написание драйверов в Vista — UMDF

В этой главе мы рассмотрим основы написания драйверов пользовательского режима в Vista — с помощью UMDF.

UMDF предоставляет возможность написания драйверов для следующих типов устройств:

- ❒ портативные устройства;
- ❒ портативные плееры;
- ❒ устройства, использующие USB;
- ❒ дополнительные устройства отображения.

Каковы главные достоинства UMDF?

- ❒ Интеграция установки и администрирования вышеперечисленных устройств со стандартными системными средствами (такими как, например, Plug and Play).
- ❒ Улучшенные стабильность, безопасность и простота написания драйверов.
- ❒ Простое и доступное использование API системы, языка C++, а также наличие автоматической генерации кода.

Прежде чем приступить непосредственно к разработке драйверов пользовательского режима в Vista, необходимо хорошо разобраться в I/O (вводе/выводе) модели Windows, а также в технологии COM.

Каждая ОС имеет свою I/O-модель. Последняя необходима ей для того, чтобы обмениваться данными с периферийными устройствами. Вот главные особенности I/O-модели Windows:

- ❒ поддержка асинхронного ввода/вывода;
- ❒ менеджер ввода/вывода предоставляет удобный интерфейс всем драйверам режима ядра. Все драйверные запросы ввода/вывода пересылаются как IRP-пакеты;

- операции ввода/вывода "слоированы";
- менеджер ввода/вывода предоставляет набор основных функций, в число которых входят и необходимые, и некоторые дополнительные функции;
- так же, как и сама ОС, драйверы имеют объектно-ориентированную архитектуру. Драйверы, устройства, которые они представляют и проч. — все так же представлено в виде объектов;
- менеджер ввода/вывода и другие компоненты системы экспортируют специальные функции режима ядра, которые вызываются драйверами для управления соответствующими объектами.

Рассмотрим, как данные "ходят" между конечным пользователем, подсистемой и менеджером ввода/вывода. Что такое подсистема? Защищенная подсистема скрывает драйверы режима ядра от конечных пользователей.

Итак:

- пользователь работает с приложением, которое с помощью запросов ввода/вывода обращается к подсистеме;
- подсистема "сортирует" запросы и отправляет их системному сервису ввода/вывода, находящемуся в режиме ядра.

Системный сервис ввода/вывода непосредственно связан с менеджером ввода/вывода, и через них запросы уже отправляются к драйверам режима ядра. В режиме ядра он оперирует различными файловыми объектами, представляющими различные же устройства — мышь, клавиатура, порты и т. д.

Теперь рассмотрим для примера, как происходит открытие файлового объекта:

1. Подсистема отправляет запрос системному сервису ввода/вывода на открытие нужного файла.
2. Менеджер ввода/вывода вызывает менеджер объектов для преобразования имени файлового объекта и для преобразования всех символьных ссылок для этого объекта.
3. Если нужный раздел не подключен, то менеджер ввода/вывода на время приостанавливает обработку запроса и начинает опрашивать имеющиеся файловые системы до тех пор, пока одна из них не сигнализирует о наличии нужного объекта на устройстве хранения данных, используемом системой. Тогда она выполняет подключение необходимого тома, и менеджер может продолжить обработку запроса.
4. Менеджер ввода/вывода выделяет необходимую память, а также инициализирует IRP для запроса.

5. Менеджер ввода/вывода вызывает драйвер файловой системы и передает ему IRP.
6. Драйвер файловой системы определяет, какой в точности запрос должен быть выполнен, проверяет его параметры, выясняет, есть ли требуемый файл в кэше, и, если нет, перенаправляет запрос более низкому драйверу в стеке.

Это — примерный сценарий, в котором может быть очень много вариаций. Тем не менее, он позволяет получить представление о том, как же работает система ввода/вывода операционной системы Windows.

А теперь перейдем к рассмотрению архитектуры самого UMDF-framework.

UMDF содержит компоненты как в пользовательском режиме, так и в режиме ядра.

В режиме ядра находятся:

- драйверы режима ядра;
- I/O-менеджер режима ядра Windows;
- фильтр-драйвер WDM режима ядра, находящийся на самом верху стека устройств (режима ядра), для каждого устройства, управляемого UMDF-драйвером. Этот фильтр-драйвер управляет обменом данными между компонентами пользовательского режима и режима ядра.

В пользовательском режиме находятся:

- менеджер драйвера;
- приложение;
- хост-процесс драйвера, включающий в себя:
  - UMDF;
  - драйвер пользовательского режима;
  - окружение времени выполнения.

Думаю, что здесь все ясно.

Наконец, подойдем "вплотную" к драйверу. Что обязан делать любой UMDF-драйвер? Три вещи:

- экспортировать DllMain как основную точку входа;
- реализовывать интерфейс `IDriverEntry` для класса драйвера;
- экспортировать `DllGetClassObject`, который обязан возвращать указатель на интерфейс `IClassFactory`, создающий экземпляр объекта обратного вызова драйвера.

Рассмотрим... как бы его лучше назвать... ну да, "жизненный цикл" UMDF-драйвера:

1. Менеджер драйвера создает хост-процесс драйвера и загружает его библиотеку путем вызова DllMain. На этом этапе происходят все глобальные инициализации.
2. UMDF создает объект драйвера и вызывает `DllGetClassObject` для того, чтобы создать объект обратного вызова в драйвере. `DllGetClassObject` возвращает указатель на `IClassFactory`.
3. Вышеупомянутый `IClassFactory` создает callback-объект с помощью метода `CreateInstance`.
4. UMDF вызывает метод `OnInitialize` callback-объекта для инициализации драйвера.
5. Как только обнаружено какое-нибудь устройство драйвера, UMDF вызывает метод `OnDeviceAdd`.
6. UMDF запрашивает интерфейсы Plug and Play и очередей, которые будут использоваться для обработки запросов ввода/вывода.
7. Когда же устройство удаляется, UMDF вызывает соответствующие методы Plug and Play для этого типа устройств, удаляет объект и вызывает метод `OnDeinitialize` интерфейса `IDriverEntry` для очистки. На этом этапе возвращаются и освобождаются все занятые ресурсы, выгружаются библиотеки и завершается хост-процесс драйвера.

Итак, а теперь без дальнейших лишних слов возьмем специальный skeleton-драйвер из поставки UMDF и разберемся, что здесь к чему.

Сначала рассмотрим главный, фактически, файл в этом драйвере, содержащий реализацию главной точки входа драйвера и экспортируемых функций для обеспечения поддержки COM-интерфейса (листинг 13.1).

Листинг 13.1. Подключаемые заголовочные файлы

```
#include "internal.h"
#include "dllsup.tmh"
```

Подключаем заголовочные файлы.

```
const GUID CLSID_MyDriverCoClass = MYDRIVER_CLASS_ID; // присваиваемое
  // значение — GUID, закодированный в структурный формат. Нужно для
  // инициализации ClassId (классового идентификатора) драйвера.
```

А теперь — метод `DllMain` (листинг 13.2).

Листинг 13.2. Метод `DllMain` — главная точка входа драйвера

```
BOOL
WINAPI
DllMain(
    HINSTANCE ModuleHandle,
    DWORD Reason,
    PVOID // не используется; зарезервировано для дальнейшего
          // использования
)
```

Метод `DllMain` — главная точка входа драйвера. Аргументы: первый — указатель на DLL-бибилиотеку данного модуля, второй — для трассировки. Метод возвращает `TRUE`. В листинге 13.3 приведено тело метода `DllMain`.

Листинг 13.3. Тело метода `DllMain`

```
{
  if (DLL_PROCESS_ATTACH == Reason) // если все успешно
  {

    WPP_INIT_TRACING(MYDRIVER_TRACING_ID); // инициализируем
      // трассировку; параметр – строка, передаваемая WPP
      // (препроцессор трассировки); для данного драйвера
      // эта строка выглядит как L"Microsoft\\UMDF\\Skeleton"

  }
  else if (DLL_PROCESS_DETACH == Reason) // если нет
  {
    WPP_CLEANUP(); // выгружаем трассировку
  }

  return TRUE; // возвращаем TRUE
}
__control_entrypoint(DllExport)
```

А в листинге 13.4 представлен заголовок метода `DllGetClassObject`.

Листинг 13.4. Метод `DllGetClassObject`

```
HRESULT
STDAPICALLTYPE
DllGetClassObject(
    __in REFCLSID ClassId,
    __in REFIID InterfaceId,
    __deref_out LPVOID *Interface
)
```

Метод создает экземпляр класса драйвера, т. к. это минимальные требования, необходимые для поддержки UMDF. Аргументы: первый — CLSID, второй — интерфейс данного объекта, нужный вызывающему, третий — местоположение для хранения указателя на интерфейс. Метод возвращает статус успешного завершения (status OK), или же идентификатор ошибки.

А теперь метод `DllGetClassObject` более подробно (листинг 13.5).

Листинг 13.5. Тело метода `DllGetClassObject`

```
{
  PCClassFactory factory; // объект ClassFactory

  HRESULT hr = S_OK;       // присваиваем переменной результата
                           // сразу status OK

  *Interface = NULL; // присваиваем указателю на интерфейс - NULL

  if (IsEqualCLSID(ClassId, CLSID_MyDriverCoClass) == false)
    // проверяем соответствие CLSID нашему параметру "coclass",
    // определенного в IDL-файле (содержащем определения
    // интерфейсов и библиотек типов)
  {
    Trace(TRACE_LEVEL_ERROR,
          L"ERROR: Called to create instance of unrecognized class
(%!GUID!)",
```

```
          &ClassId
          );

    return CLASS_E_CLASSNOTAVAILABLE;
  } // не совпало; мы выводим трассировочное сообщение
    // и возвращаем ошибку

  factory = new CClassFactory(); // создаем экземпляр класса
                                 // вызывающему

  if (NULL == factory)  // если не удалось
  {
    hr = E_OUTOFMEMORY; // сигнализируем выход за пределы памяти
  }

  if (SUCCEEDED(hr)) // проверяем результат
  {
    // запрашиваем интерфейс
    hr = factory->QueryInterface(InterfaceId, Interface);
    factory->Release(); // выполняем очистку
  }

  return hr; // возвращаем результат
}
```

Далее рассмотрим реализацию callback-объекта драйвера в коде.

С помощью кода из листинга 13.6 мы просто подключаем заголовочные файлы.

Листинг 13.6. Подключаемые заголовочные файлы

```
#include "internal.h"
#include "driver.tmh"
```

Начнем с метода `CreateInstance` (листинг 13.7).

Листинг 13.7. Метод `CreateInstance`

```
HRESULT
CMyDriver::CreateInstance(
  __out PCMyDriver *Driver
)
```

Ранее мы уже говорили об этом статическом методе. Единственный аргумент, который он принимает — месторасположение указателя на новый экземпляр. Возвращает этот метод `S_OK` (status OK) в случае успеха и ошибку в противном случае.

Теперь метод `CreateInstance` подробнее (листинг 13.8).

Листинг 13.8. Создание и инициализация callback-объекта (метод `CreateInstance`)

```
{
  PCMyDriver driver; // создаем переменную callback-объекта драйвера
  HRESULT hr;        // создаем переменную для хранения результата
  driver = new CMyDriver(); // выделяем память для объекта драйвера

  if (NULL == driver)       // если выделение памяти не удалось...
  {
    return E_OUTOFMEMORY;   // ... сигнализируем выход за пределы
                            // памяти
  }

  hr = driver->Initialize(); // инициализируем callback-объект
                             // с помощью метода Initialize, который
                             // подробнее мы рассмотрим далее

  if (SUCCEEDED(hr)) // если инициализация успешна...
  {
    *Driver = driver; // то мы сохраняем указатель на новый объект,
            // используя переданный методу CreateInstance-аргумент
  }
```

```
  else      // если же ничего не вышло...
  {
    driver->Release(); // ... удаляем созданную ссылку на объект
  }

  return hr;
}
```

Рассмотрим код метода `Initialize` (листинг 13.9).

Листинг 13.9. Метод `Initialize`

```
HRESULT
CMyDriver::Initialize(
    VOID
)
{
  return S_OK;
}
```

Этот метод вызывается для инициализации callback-объекта. Не принимает аргументов и не возвращает значения.

Далее — метод `QueryInterface` (листинги 13.10 и 13.11).

Листинг 13.10. Метод `QueryInterface`

```
HRESULT
CMyDriver::QueryInterface(
  __in REFIID InterfaceId,
  __out PVOID *Interface
)
```

Этот метод возвращает указатель на запрошенный интерфейс callback-объекта. Метод `QueryInterface` принимает два аргумента: `InterfaceId` — здесь передается идентификатор нужного интерфейса, `Interface` — это местоположение для хранения указателя. Метод возвращает статус успешного завершения в случае успеха и `E_NOINTERFACE` в случае неудачи (сигнализирует об отсутствии интерфейса).

Листинг 13.11. Тело метода `QueryInterface`

```
{
  // если переданный ID и ID интерфейса драйвера совпадают...
  if (IsEqualIID(InterfaceId, __uuidof(IDriverEntry)))
  {
    *Interface = QueryIDriverEntry(); // ... заполняем переданный
                                      // указатель
    return S_OK;          // и возвращаем status OK
  }
  else // в противном случае...
  {
    return CUnknown::QueryInterface(InterfaceId, Interface);
    // ... сообщаем о неудаче
  }
}
```

Теперь — метод `OnDeviceAdd` (листинги 13.12 и 13.13).

Листинг 13.12. Метод `OnDeviceAdd`

```
HRESULT
CMyDriver::OnDeviceAdd(
  __in IWDFDriver *FxWdfDriver,
  __in IWDFDeviceInitialize *FxDeviceInit
)
```

Об этом методе мы говорили ранее. Принимает два аргумента — объект драйвера и информацию для инициализации. Возвращает статус проделанной операции.

Листинг 13.13. Тело метода `OnDeviceAdd`

```
{
  HRESULT hr; // переменная для хранения результата

  PCMyDevice device = NULL; // переменная для хранения объекта устройства
```

```
  hr = CMyDevice::CreateInstance(FxWdfDriver, FxDeviceInit, &device);
  // создаем новый экземпляр callback-объекта драйвера

  if (SUCCEEDED(hr))   // если операция завершилась успешно...
  {
    hr = device->Configure(); // ... производим конфигурирование
                              // устройства
  }

  if (NULL != device)  // если же нет...
  {
    device->Release(); // выполняем очистку...
  }

  return hr; // и в любом случае возвращаем результат
}
```

А теперь рассмотрим создание callback-объекта устройства.

Для начала мы просто подключаем заголовочные файлы (листинг 13.14).

Листинг 13.14. Подключаемые заголовочные файлы

```
#include "internal.h"
#include "device.tmh"
```

Теперь — метод `CreateInstance` (листинги 13.15 и 13.16).

Листинг 13.15. Метод `CreateInstance`

```
HRESULT
CMyDevice::CreateInstance (
  __in IWDFDriver   *FxDriver,
  __in IWDFDeviceInitialize * FxDeviceInit,
  __out PCMyDevice  *Device
)
```

Ранее мы уже говорили об этом методе. Аргументы, которые он принимает: первый — месторасположение указателя на объект драйвера, второй — параметры устройства, третий — месторасположение указателя на объект устройства. Метод возвращает статус проведенной операции.

Листинг 13.16. Создание и инициализация callback-объекта драйвера (метод `CreateInstance`)

```
{
  PCMyDevice device; // переменная для хранения объекта устройства
  HRESULT hr;        // переменная для хранения результата

  device = new CMyDevice(); // выделяем память для объекта устройства

  if (NULL == device)       // если выделение памяти не удалось...
  {
    // ... сигнализируем выход за пределы памяти
    return E_OUTOFMEMORY;
  }

  hr = device->Initialize(FxDriver, FxDeviceInit);
  // инициализируем объект устройства

  if (SUCCEEDED(hr))  // если удалось...
  {
    *Device = device; // возвращаем указатель на созданный объект
  }
  else  // если же нет...
  {
    device->Release(); // ... выполняем очистку
  }

  return hr; // возвращаем результат
}
```

Далее — метод `Initialize` (листинги 13.17 и 13.18).

Листинг 13.17. Метод `Initialize`

```
HRESULT
CMyDevice::Initialize(
  __in IWDFDriver           * FxDriver,
  __in IWDFDeviceInitialize * FxDeviceInit
)
```

Этот метод вызывается для инициализации callback-объекта и создания соответствующего объекта устройства. Аргументы все знакомые. Возвращает статус.

Листинг 13.18. Тело метода `Initialize`

```
{
  IWDFDevice *fxDevice; // указатель устройства
  HRESULT hr;           // результат

  // установка блокирования
  FxDeviceInit->SetLockingConstraint(WdfDeviceLevel);

  FxDeviceInit->SetFilter(); // установка фильтр-драйвера;
                             // эта строчка нужна
    // ТОЛЬКО в том случае, если у вас есть фильтр-драйвер

  {
    IUnknown *unknown = this->QueryIUnknown();
      // ссылка на интерфейс IUnknown,
      // который должны поддерживать абсолютно все драйверы;
      // он реализует такие методы, как Release и AddRef

    hr = FxDriver->CreateDevice (FxDeviceInit, unknown, &fxDevice);
      // создание устройства

    unknown->Release (); // очистка
  }
```

```
  if (SUCCEEDED (hr))  // если успешно...
  {
    m_FxDevice = fxDevice; // устанавливаем значение FxDevice
    fxDevice->Release();   // и очищаем более ненужное
  }

  return hr; // возвращаем результат
}
```

Рассмотрим метод `Configure` (листинги 13.19 и 13.20).

Листинг 13.19. Метод `Configure`

```
HRESULT
CMyDevice::Configure(
  VOID
)
```

Этот метод вызывается после инициализации callback-объекта устройства. Метод устанавливает очереди устройства и соответствующие им callback-объекты. Принимает объект устройства и возвращает статус.

Листинг 13.20. Тело метода `Configure`

```
{
  return S_OK; // возвращаем status OK
}
```

Рассмотрим метод `QueryInterface` (листинги 13.21 и 13.22).

Листинг 13.21. Метод `QueryInterface`

```
HRESULT
CMyDevice::QueryInterface(
  __in REFIID InterfaceId,
  __out PVOID *Object
)
```

Метод узнает указатель на запрошенный интерфейс callback-объекта. Аргументы знакомые. Метод возвращает статус успешного завершения или `E_NOINTERFACE`.

Листинг 13.22. Тело метода `QueryInterface`

```
{
  // собственно говоря, запрашиваем указатель на нужный интерфейс
  return CUnknown::QueryInterface(InterfaceId, Object);
}
```

Еще два наиболее значительных файла (впрочем, там все значительные) Comsup.h и Comsup.cpp — файлы для поддержки COM — нам здесь уже не так интересны и только помешают ясному пониманию концепции драйвера.

Ну что тут еще сказать? Скажу лишь, что inf-файлы отныне являются inx-файлами.

Приложения

Приложение 1

Краткий словарь терминов

ACPI

Advanced Configuration and Power Interface. Интерфейс, определяющий механизмы управления энергопотреблением и конфигурирования аппаратного обеспечения и компонентов операционной системы.

Callback, callback function

Процедура обратного вызова. Обратный вызов означает, что какая-либо функция сообщает другой, чтобы он вызвал указанную функцию в нужный момент.

CDECL

Один из типов CC (calling convention — соглашение о вызовах), являющихся стандартным для программ на языках C и C++. Аргументы передаются через стек, справа налево, вызванная функция забирает аргументы из стека. Отличается от STDCALL, прежде всего, тем, что генерирует исполняемые файлы заметно большего размера. Это происходит оттого, что в CDECL любой взвывающий код должен включать в себя функциональность по очистке стека.

Class Driver

Классовый драйвер. Высокоуровневый драйвер, который представляет поддержку целого класса устройств.

Deferred Procedure Call

Процедура отложенного вызова. Функция, которая будет вызвана позже. Управляет ими (а точнее, их очередью) I/O-диспетчер (можно считать, "в более спокойной обстановке").

Device Extension

Расширение объекта устройства. Структура, определяемая автором драйвера.

Device Instance

Физический аппаратный компонент. Экземпляр устройства, обслуживаемого данным драйвером.

Device Stack, Driver Stack

Стек устройств, стек драйверов.

Все устройства образуют так называемое "дерево" устройств, которое можно увидеть, если запустить программу DeviceTree (входит в DDK).

Стек устройств формируется динамически по мере загрузки драйверов и подключения устройств.

DIRQL

Уровни аппаратных прерываний.

Dispatch Routines

Рабочие процедуры, процедуры обработки. Все их драйвер регистрирует во время работы своей функции — точки входа.

DMA, Direct Memory Access

Метод обмена данными между устройством и оперативной памятью без участия центрального процессора.

DpcForISR, Deferred Procedure Call for Interrupt Service Routine

Процедура отложенного вызова для обслуживания прерываний.

FASTCALL

Один из типов CC (calling convention — соглашение о вызовах). Характеризуется тем, что первые два аргумента (типа `DWORD` или еще меньше) передаются через регистры `ECX` и `EDX`. Остальные аргументы передаются через стек справа налево. Вызванная функция забирает аргументы из стека.

Filter Device Object

Объект устройства, создаваемый фильтр-драйвером.

Filter Driver

Фильтр-драйвер. Драйвер, предназначенный для выполнения дополнительных манипуляций над IRP-пакетами основного драйвера (вплоть до того, что самостоятельно отвергает их), к которому он подключается в стеке либо сверху (Upper Filter), либо снизу (Lower Filter). У одного основного драйвера может быть несколько фильтр-драйверов. Все фильтр-драйверы являются для драйвера "прозрачными".

GDT

Global Descriptor Table — глобальная таблица дескрипторов. Может быть в системе только одна. Все программы и задачи системы хранят в этой таблице свои дескрипторы.

HAL, Hardware Abstraction Layer

Слой аппаратных абстракций, скрывающий от более высокоуровневых компонентов ОС специфику конкретного аппаратного обеспечения.

IDT

Interrupt Descriptor Table — системная таблица, содержащая дескрипторы обработчиков прерываний (в том числе и адреса этих прерываний). Дескрипторы также называются шлюзами.

Сама таблица размещена в защищенной области памяти режима ядра.

Как система работает с этой таблицей? Как только происходит прерывание (или его регистрация), процессор по его номеру осуществляет поиск в таблице IDT. В соответствии с этим номером установлен шлюз, с помощью которого процессор узнает адреса обработчика данного прерывания и передает ему управление.

IOCTL

I/O ConTroL code — код управления вводом/выводом. Используется для обращения к драйверу с различными запросами. Программист может и сам создавать собственные IOCTL-коды.

I/O Manager

Менеджер ввода/вывода. Код, работающий в режиме ядра, управляющий вводом/выводом. Занимается он, понятно, среди прочего, и вопросами ввода/вывода драйвера (например, его "общением" с пользовательским приложением).

IRP, Input/output Request Packet, IRP request, IRP packet

Пакет запроса на ввод/вывод.

IRQL, Interrupt ReQuest Level

Уровень приоритета выполнения.

ISR, Interrupt Service Routine

Процедура обслуживания прерываний. Функция, которая вызывается в момент генерации прерывания устройством, которое обслуживает драйвер.

KMDF

Новый Framework от Microsoft для написания драйверов режима ядра. Входит в новую драйверную платформу WDF.

В книге рассматриваются архитектура KMDF и написание драйверов с применением этой среды.

Layering

Многослойность. Поддерживаемая моделью WDM возможность реализовывать стековое соединение между драйверами.

Legacy Driver, NT Style Driver

Драйвер "в стиле" NT. Драйвер, не поддерживаемый WDM и не работающий с Plug and Play.

Major IRP Code

Основной код IRP-пакета. Число, указывающее назначение IRP-пакета.

Minidriver

Мини-драйвер. Меньше обычного драйвера. Обслуживает, главным образом, специфичные для данного аппаратного обеспечения возможности.

Minor IRP Code

Младший код IRP-пакета. Уточняет старший код.

Monolithic Driver

Монолитный драйвер. Драйвер, не включенный в стек устройств и полностью самостоятельно обрабатывающий свои IRP-пакеты.

Mutex

Разновидность семафора. Используется для синхронизации одновременно выполняющихся потоков. Все потоки, пытающиеся обратиться к данным, на которые какой-либо другой поток установил мьютекс, "уснут" до его снятия.

PnP Manager, Plug and Play Manager

PnP-менеджер. Один из ключевых компонентов операционной системы, конструктивно состоящий из двух частей: PnP-менеджера, работающего в режиме ядра, и PnP-менеджера пользовательского режима. Осуществляют работу и поддержку механизма Plug and Play.

Port Driver

Порт-драйвер. Самый "низкостоящий" драйвер, который отвечает на стандартные системные запросы. Изолирует вышестоящие компоненты от особенностей аппаратного обеспечения. Часто работает в тандеме с мини-драйвером.

RPC (Remote Procedure Call)

Протокол вызова удаленных процедур. Концепция этого протокола проста, привлекательна и довольно удобна.

RPC обеспечивает "прозрачную" передачу данных по сети, позволяющую приложению вызывать процедуру на каком-либо удаленном компьютере, при этом не заботясь о протоколах сети и прочем. RPC обеспечивает это с помощью особой двухуровневой процедуры.

Существует особая программа — транслятор портов (portmapper). Транслятор портов регистрирует на сервере (понятно, что архитектура RPC — клиент-серверная) все приложения, которые предоставляют свои услуги RPC. Portmapper попросту устанавливает соответствие между номером процедуры RPC и номером TCP-порта (UDP-порта), на котором "висит" приложение и ожидает запросов. Приложение, ранее зарегистрировавшееся в Portmapper, уже сообщило ему все данные (номера портов, номер версии, номера процедур и т. д.). Клиент просто отправляет запрос транслятору портов (который — стандартно — находится на порту 111), а тот ему сообщает, куда (на какой порт) отправить свой запрос.

STDCALL

Один из типов CC (calling convention — концепция вызова). Характеризуется тем, что вызывающая функция "кладет" параметры, передаваемые подпрограмме, в стек справа налево. Аргументы по умолчанию передаются по значению. Вызванная функция "забирает" свои параметры из стека.

Symbolic Link

Символьная ссылка. В нашем случае представляет собой файловый объект с особыми свойствами, позволяющий получать доступ к файлу/объекту, с ним ассоциированному.

Synchronization Objects

Объекты синхронизации. Эти объекты позволяют программным потокам синхронизировать свой доступ к тем или иным совместно используемым данным.

В эту категорию входят:

- ❒ события;
- ❒ потоки;
- ❒ спин-блокировки;
- ❒ семафоры;
- ❒ мьютексы.

UMDF

Новый Framework от Microsoft для разработки драйверов пользовательского режима. Входит в новую драйверную платформу WDF.

В книге рассматриваются архитектура UMDF и написание драйверов с использованием этой среды.

User Space

Пользовательское пространство памяти. Область памяти, выделенная для пользовательских приложений.

WDF

Новая драйверная платформа Microsoft. Предназначена, главным образом, для новой платформы Windows, но позволяет разрабатывать драйверы и для XP.

WDF состоит из двух частей: KMDF (для программирования драйверов режима ядра) и UMDF (для программирования драйверов пользовательского режима).

Все три понятия — WDF, KMDF, UMDF — подробно разбираются на страницах книги.

WDM Streaming Architecture

Потоковая архитектура WDM. WDM Streaming — это решение уровня ядра. Решает проблемы синхронизации рендеринга больших объемов данных практически в реальном времени (и в Windows 98, и в Windows NT). Основная цель этой технологии — обеспечить как можно лучшую работу в реальном времени для медиапотоков.

Приложение 2

Полезные исходные коды из DDK

В этом приложении содержатся исходные коды из DDK, необходимые/полезные при изучении материала, представленного в книге, и иллюстрирующие рассматривающиеся в ней основные типы драйверов. Используемая версия продукта — Windows NT/2000 Driver Development Kit. Тексты файлов приводятся без каких-либо изменений, "купюр" и с полным сохранением указаний авторских прав.

П2.1. Исходные коды монитора порта принтера

Здесь приводится листинг главного файла в мониторе порта принтера, написание которого рассматривалось на страницах этой книги.

Листинг П2.1. Файл localmon.c

```
/*++
Copyright (c) 1990-1998 Microsoft Corporation
All rights reserved
Module Name:
localmon.c
--*/
#include "precomp.h"
#pragma hdrstop
#include "lmon.h"
#include "irda.h"
```

```
HANDLE LcmhMonitor;
HANDLE LcmhInst;
CRITICAL_SECTION LcmSpoolerSection;
DWORD LocalmonDebug;
DWORD LcmPortInfo1Strings[]={FIELD_OFFSET(PORT_INFO_1, pName),
(DWORD)-1};
DWORD LcmPortInfo2Strings[]={FIELD_OFFSET(PORT_INFO_2, pPortName),
FIELD_OFFSET(PORT_INFO_2, pMonitorName),
FIELD_OFFSET(PORT_INFO_2, pDescription),
(DWORD)-1};
WCHAR szPorts[] = L"ports";
WCHAR szPortsEx[] = L"portsex"; /* Extra ports values */
WCHAR szFILE[] = L"FILE:";
WCHAR szLcmCOM[] = L"COM";
WCHAR szLcmLPT[] = L"LPT";
WCHAR szIRDA[] = L"IR";
extern WCHAR szWindows[];
extern WCHAR szINIKey_TransmissionRetryTimeout[];
BOOL
LocalMonInit(HANDLE hModule)
{
LcmhInst = hModule;
InitializeCriticalSection(&LcmSpoolerSection);
return TRUE;
}
BOOL
LcmEnumPorts(
HANDLE hMonitor,
LPWSTR pName,
DWORD Level,
LPBYTE pPorts,
DWORD cbBuf,
LPDWORD pcbNeeded,
LPDWORD pcReturned
)
```

```
{
PINILOCALMON pIniLocalMon = (PINILOCALMON)hMonitor;
PINIPORT pIniPort;
DWORD cb;
LPBYTE pEnd;
DWORD LastError=0;
LcmEnterSplSem();
cb=0;
pIniPort = pIniLocalMon->pIniPort;
CheckAndAddIrdaPort(pIniLocalMon);
while (pIniPort) {
if ( ! (pIniPort->Status & PP_FILEPORT) ) {
cb+=GetPortSize(pIniPort, Level);
}
pIniPort=pIniPort->pNext;
}
*pcbNeeded=cb;
if (cb <= cbBuf) {
pEnd=pPorts+cbBuf;
*pcReturned=0;
pIniPort = pIniLocalMon->pIniPort;
while (pIniPort) {
if (!(pIniPort->Status & PP_FILEPORT)) {
pEnd = CopyIniPortToPort(pIniPort, Level, pPorts, pEnd);
if( !pEnd ){
LastError = GetLastError();
break;
}
switch (Level) {
case 1:
pPorts+=sizeof(PORT_INFO_1);
break;
case 2:
pPorts+=sizeof(PORT_INFO_2);
break;
```

```
default:
DBGMSG(DBG_ERROR,
("EnumPorts: invalid level %d", Level));
LastError = ERROR_INVALID_LEVEL;
goto Cleanup;
}
(*pcReturned)++;
}
pIniPort=pIniPort->pNext;
}
} else
LastError = ERROR_INSUFFICIENT_BUFFER;
Cleanup:
LcmLeaveSplSem();
if (LastError) {
SetLastError(LastError);
return FALSE;
} else
return TRUE;
}
BOOL
LcmxEnumPorts(
LPWSTR pName,
DWORD Level,
LPBYTE pPorts,
DWORD cbBuf,
LPDWORD pcbNeeded,
LPDWORD pcReturned
)
{
return LcmEnumPorts(LcmhMonitor, pName, Level, pPorts, cbBuf, pcbNeeded,
pcReturned);
}
BOOL
LcmOpenPort(
HANDLE hMonitor,
```

```
LPWSTR pName,
PHANDLE pHandle
)
{
PINILOCALMON pIniLocalMon = (PINILOCALMON)hMonitor;
PINIPORT pIniPort;
BOOL bRet = FALSE;
LcmEnterSplSem();
if ( IS_FILE_PORT(pName) ) {
//
// We will always create multiple file port
// entries, so that the spooler can print
// to multiple files.
//
DBGMSG(DBG_TRACE, ("Creating a new pIniPort for %ws\n", pName));
pIniPort = LcmCreatePortEntry( pIniLocalMon, pName );
if ( !pIniPort )
goto Done;
pIniPort->Status |= PP_FILEPORT;
*pHandle = pIniPort;
bRet = TRUE;
goto Done;
}
pIniPort = FindPort(pIniLocalMon, pName);
if ( !pIniPort )
goto Done;
//
// For LPT ports language monitors could do reads outside Start/End doc
// port to do bidi even when there are no jobs printing. So we do a
// CreateFile and keep the handle open all the time.
//
// But for COM ports you could have multiple devices attached to a COM
// port (ex. a printer and some other device with a switch)
// To be able to use the other device they write a utility which will
// do a net stop serial and then use the other device. To be able to
```

```
// stop the serial service spooler should not have a handle to the port.
// So we need to keep handle to COM port open only when there is a job
// printing
//
//
if ( IS_COM_PORT(pName) ) {
bRet = TRUE;
goto Done;
}
//
// If it is not a port redirected we are done (succeed the call)
//
if ( ValidateDosDevicePort(pIniPort) ) {
bRet = TRUE;
//
// If it isn't a true dosdevice port (ex. net use lpt1
\\<server>\printer)
// then we need to do CreateFile and CloseHandle per job so that
// StartDoc/EndDoc is issued properly for the remote printer
//
if ( (pIniPort->Status & PP_DOSDEVPORT) &&
!(pIniPort->Status & PP_COMM_PORT) ) {
CloseHandle(pIniPort->hFile);
pIniPort->hFile = INVALID_HANDLE_VALUE;
(VOID)RemoveDosDeviceDefinition(pIniPort);
}
}
Done:
if ( !bRet && pIniPort && (pIniPort->Status & PP_FILEPORT) )
DeletePortNode(pIniLocalMon, pIniPort);
if ( bRet )
*pHandle = pIniPort;
LcmLeaveSplSem();
return bRet;
}
```

```
BOOL
LcmxOpenPort(
LPWSTR pName,
PHANDLE pHandle
)
{
return LcmOpenPort(LcmhMonitor, pName, pHandle);
}
BOOL
LcmStartDocPort(
HANDLE hPort,
LPWSTR pPrinterName,
DWORD JobId,
DWORD Level,
LPBYTE pDocInfo)
{
PINIPORT pIniPort = (PINIPORT)hPort;
HANDLE hToken;
PDOC_INFO_1 pDocInfo1 = (PDOC_INFO_1)pDocInfo;
DWORD Error = 0;
DBGMSG(DBG_TRACE, ("StartDocPort(%08x, %ws, %d, %d, %08x)\n",
hPort, pPrinterName, JobId, Level, pDocInfo));
if (pIniPort->Status & PP_STARTDOC) {
return TRUE;
}
LcmEnterSplSem();
pIniPort->Status |= PP_STARTDOC;
LcmLeaveSplSem();
pIniPort->hPrinter = NULL;
pIniPort->pPrinterName = AllocSplStr(pPrinterName);
if (pIniPort->pPrinterName) {
if (OpenPrinter(pPrinterName, &pIniPort->hPrinter, NULL)) {
pIniPort->JobId = JobId;
//
// For COMx port we need to validates dos device now since
```

```
// we do not do it during OpenPort
//
if ( IS_COM_PORT(pIniPort->pName) &&
!ValidateDosDevicePort(pIniPort) ) {
goto Fail;
}
if ( IS_FILE_PORT(pIniPort->pName) ) {
HANDLE hFile = INVALID_HANDLE_VALUE;
if (pDocInfo1 &&
pDocInfo1->pOutputFile &&
pDocInfo1->pOutputFile[0] ){
hFile = CreateFile( pDocInfo1->pOutputFile,
GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL,
OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN,
NULL );
DBGMSG(DBG_TRACE,
("Print to file and the handle is %x\n", hFile));
} else {
HANDLE hToken;
INT_PTR rc;
hToken = RevertToPrinterSelf();
rc = DialogBoxParam( LcmhInst,
MAKEINTRESOURCE( DLG_PRINTTOFILE ),
NULL, PrintToFileDlg,
(LPARAM)&hFile );
ImpersonatePrinterClient(hToken);
if( rc == -1 ) {
goto Fail;
} else if( rc == 0 ) {
Error = ERROR_PRINT_CANCELLED;
goto Fail;
}
}
```

```
if (hFile != INVALID_HANDLE_VALUE)
SetEndOfFile(hFile);
pIniPort->hFile = hFile;
} else if ( IS_IRDA_PORT(pIniPort->pName) ) {
if ( !IrdaStartDocPort(pIniPort) )
goto Fail;
} else if ( !(pIniPort->Status & PP_DOSDEVPORT) ) {
//
// For non dosdevices CreateFile on the name of the port
//
pIniPort->hFile = CreateFile(pIniPort->pName,
GENERIC_WRITE,
FILE_SHARE_READ,
NULL,
OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_SEQUENTIAL_SCAN,
NULL);
if ( pIniPort->hFile != INVALID_HANDLE_VALUE )
SetEndOfFile(pIniPort->hFile);
} else if ( !IS_COM_PORT(pIniPort->pName) ) {
if ( !FixupDosDeviceDefinition(pIniPort) )
goto Fail;
}
}
} // end of if (pIniPort->pPrinterName)
if (pIniPort->hFile == INVALID_HANDLE_VALUE)
goto Fail;
return TRUE;
Fail:
SPLASSERT(pIniPort->hFile == INVALID_HANDLE_VALUE);
LcmEnterSplSem();
pIniPort->Status &= ~PP_STARTDOC;
LcmLeaveSplSem();
if (pIniPort->hPrinter) {
```

```
ClosePrinter(pIniPort->hPrinter);
}
if (pIniPort->pPrinterName) {
FreeSplStr(pIniPort->pPrinterName);
}
if (Error)
SetLastError(Error);
return FALSE;
}
BOOL
LcmWritePort(
HANDLE hPort,
LPBYTE pBuffer,
DWORD cbBuf,
LPDWORD pcbWritten)
{
PINIPORT pIniPort = (PINIPORT)hPort;
BOOL rc;
DBGMSG(DBG_TRACE, ("WritePort(%08x, %08x, %d)\n", hPort, pBuffer,
cbBuf));
if ( IS_IRDA_PORT(pIniPort->pName) )
rc = IrdaWritePort(pIniPort, pBuffer, cbBuf, pcbWritten);
else if ( !pIniPort->hFile || pIniPort->hFile == INVALID_HANDLE_VALUE ) {
SetLastError(ERROR_INVALID_HANDLE);
return FALSE;
} else {
rc = WriteFile(pIniPort->hFile, pBuffer, cbBuf, pcbWritten, NULL);
if ( rc && *pcbWritten == 0 ) {
SetLastError(ERROR_TIMEOUT);
rc = FALSE;
}
}
DBGMSG(DBG_TRACE, ("WritePort returns %d; %d bytes written\n", rc,
*pcbWritten));
return rc;
}
```

```
BOOL
LcmReadPort(
HANDLE hPort,
LPBYTE pBuffer,
DWORD cbBuf,
LPDWORD pcbRead)
{
PINIPORT pIniPort = (PINIPORT)hPort;
BOOL rc;
DBGMSG(DBG_TRACE, ("ReadPort(%08x, %08x, %d)\n", hPort, pBuffer, cbBuf));
if ( !pIniPort->hFile ||
pIniPort->hFile == INVALID_HANDLE_VALUE ||
!(pIniPort->Status & PP_COMM_PORT) ) {
SetLastError(ERROR_INVALID_HANDLE);
return FALSE;
}
rc = ReadFile(pIniPort->hFile, pBuffer, cbBuf, pcbRead, NULL);
DBGMSG(DBG_TRACE, ("ReadPort returns %d; %d bytes read\n", rc,
*pcbRead));
return rc;
}
BOOL
LcmEndDocPort(
HANDLE hPort
)
{
PINIPORT pIniPort = (PINIPORT)hPort;
DBGMSG(DBG_TRACE, ("EndDocPort(%08x)\n", hPort));
if (!(pIniPort->Status & PP_STARTDOC)) {
return TRUE;
}
// The flush here is done to make sure any cached IO's get written
// before the handle is closed. This is particularly a problem
// for Intelligent buffered serial devices
FlushFileBuffers(pIniPort->hFile);
```

```
//
// For any ports other than real LPT ports we open during StartDocPort
// and close it during EndDocPort
//
if ( !(pIniPort->Status & PP_COMM_PORT) || IS_COM_PORT(pIniPort->pName) )
{
if ( IS_IRDA_PORT(pIniPort->pName) ) {
IrdaEndDocPort(pIniPort);
} else {
CloseHandle(pIniPort->hFile);
pIniPort->hFile = INVALID_HANDLE_VALUE;
if ( pIniPort->Status & PP_DOSDEVPORT ) {
(VOID)RemoveDosDeviceDefinition(pIniPort);
}
if ( IS_COM_PORT(pIniPort->pName) ) {
pIniPort->Status &= ~(PP_COMM_PORT | PP_DOSDEVPORT);
FreeSplStr(pIniPort->pDeviceName);
pIniPort->pDeviceName = NULL;
}
}
}
SetJob(pIniPort->hPrinter, pIniPort->JobId, 0, NULL,
JOB_CONTROL_SENT_TO_PRINTER);
ClosePrinter(pIniPort->hPrinter);
FreeSplStr(pIniPort->pPrinterName);
//
// Startdoc no longer active.
//
pIniPort->Status &= ~PP_STARTDOC;
return TRUE;
}
BOOL
LcmClosePort(
HANDLE hPort
)
```

```
{
PINIPORT pIniPort = (PINIPORT)hPort;
FreeSplStr(pIniPort->pDeviceName);
pIniPort->pDeviceName = NULL;
if (pIniPort->Status & PP_FILEPORT) {
LcmEnterSplSem();
DeletePortNode(pIniPort->pIniLocalMon, pIniPort);
LcmLeaveSplSem();
} else if ( pIniPort->Status & PP_COMM_PORT ) {
(VOID) RemoveDosDeviceDefinition(pIniPort);
if ( pIniPort->hFile != INVALID_HANDLE_VALUE ) {
CloseHandle(pIniPort->hFile);
pIniPort->hFile = INVALID_HANDLE_VALUE;
}
pIniPort->Status &= ~(PP_COMM_PORT | PP_DOSDEVPORT);
}
return TRUE;
}
BOOL
LcmAddPortEx(
HANDLE hMonitor,
LPWSTR pName,
DWORD Level,
LPBYTE pBuffer,
LPWSTR pMonitorName
)
{
PINILOCALMON pIniLocalMon = (PINILOCALMON)hMonitor;
LPWSTR pPortName;
DWORD Error;
LPPORT_INFO_1 pPortInfo1;
LPPORT_INFO_FF pPortInfoFF;
switch (Level) {
case (DWORD)-1:
pPortInfoFF = (LPPORT_INFO_FF)pBuffer;
```

```
pPortName = pPortInfoFF->pName;
break;
case 1:
pPortInfo1 = (LPPORT_INFO_1)pBuffer;
pPortName = pPortInfo1->pName;
break;
default:
SetLastError(ERROR_INVALID_LEVEL);
return(FALSE);
}
if (!pPortName) {
SetLastError(ERROR_INVALID_PARAMETER);
return(FALSE);
}
if (PortExists(pName, pPortName, &Error)) {
SetLastError(ERROR_INVALID_PARAMETER);
return(FALSE);
}
if (Error != NO_ERROR) {
SetLastError(Error);
return(FALSE);
}
if (!LcmCreatePortEntry(pIniLocalMon, pPortName)) {
return(FALSE);
}
if (!WriteProfileString(szPorts, pPortName, L"")) {
LcmDeletePortEntry( pIniLocalMon, pPortName );
return(FALSE);
}
return TRUE;
}
BOOL
LcmxAddPortEx(
LPWSTR pName,
DWORD Level,
```

```
LPBYTE pBuffer,
LPWSTR pMonitorName
)
{
return LcmAddPortEx(LcmhMonitor, pName, Level, pBuffer, pMonitorName);
}
BOOL
LcmGetPrinterDataFromPort(
HANDLE hPort,
DWORD ControlID,
LPWSTR pValueName,
LPWSTR lpInBuffer,
DWORD cbInBuffer,
LPWSTR lpOutBuffer,
DWORD cbOutBuffer,
LPDWORD lpcbReturned)
{
PINIPORT pIniPort = (PINIPORT)hPort;
BOOL rc;
DBGMSG(DBG_TRACE,
("GetPrinterDataFromPort(%08x, %d, %ws, %ws, %d, ",
hPort, ControlID, pValueName, lpInBuffer, cbInBuffer));
if ( !ControlID ||
!pIniPort->hFile ||
pIniPort->hFile == INVALID_HANDLE_VALUE ||
!(pIniPort->Status & PP_DOSDEVPORT) ) {
SetLastError(ERROR_INVALID_PARAMETER);
return FALSE;
}
rc = DeviceIoControl(pIniPort->hFile,
ControlID,
lpInBuffer,
cbInBuffer,
lpOutBuffer,
cbOutBuffer,
```

```
lpcbReturned,
NULL);
DBGMSG(DBG_TRACE,
("%ws, %d, %d)\n", lpOutBuffer, cbOutBuffer, lpcbReturned));
return rc;
}
BOOL
LcmSetPortTimeOuts(
HANDLE hPort,
LPCOMMTIMEOUTS lpCTO,
DWORD reserved) // must be set to 0
{
PINIPORT pIniPort = (PINIPORT)hPort;
COMMTIMEOUTS cto;
if (reserved != 0)
return FALSE;
if ( !(pIniPort->Status & PP_DOSDEVPORT) ) {
SetLastError(ERROR_INVALID_PARAMETER);
return FALSE;
}
if ( GetCommTimeouts(pIniPort->hFile, &cto) )
{
cto.ReadTotalTimeoutConstant = lpCTO->ReadTotalTimeoutConstant;
cto.ReadIntervalTimeout = lpCTO->ReadIntervalTimeout;
return SetCommTimeouts(pIniPort->hFile, &cto);
}
return FALSE;
}
VOID
LcmShutdown(
HANDLE hMonitor
)
{
PINIPORT pIniPort;
PINIPORT pIniPortNext;
```

```
PINILOCALMON pIniLocalMon = (PINILOCALMON)hMonitor;
//
// Delete the ports, then delete the LOCALMONITOR.
//
for( pIniPort = pIniLocalMon->pIniPort; pIniPort; pIniPort = pIniPortNext
){
pIniPortNext = pIniPort->pNext;
FreeSplMem( pIniPort );
}
FreeSplMem( pIniLocalMon );
}
BOOL
LcmxXcvOpenPort(
LPCWSTR pszObject,
ACCESS_MASK GrantedAccess,
PHANDLE phXcv
)
{
return LcmXcvOpenPort(LcmhMonitor, pszObject, GrantedAccess, phXcv);
}
MONITOR2 Monitor2 = {
sizeof(MONITOR2),
LcmEnumPorts,
LcmOpenPort,
NULL, // OpenPortEx is not supported
LcmStartDocPort,
LcmWritePort,
LcmReadPort,
LcmEndDocPort,
LcmClosePort,
NULL, // AddPort is not supported
LcmAddPortEx,
NULL, // ConfigurePort is not supported
NULL, // DeletePort is not supported
LcmGetPrinterDataFromPort,
LcmSetPortTimeOuts,
```

```
LcmXcvOpenPort,
LcmXcvDataPort,
LcmXcvClosePort,
LcmShutdown
};
LPMONITOR2
LocalMonInitializePrintMonitor2(
PMONITORINIT pMonitorInit,
PHANDLE phMonitor
)
{
LPWSTR pPortTmp;
DWORD dwCharCount=1024, rc, i, j;
PINILOCALMON pIniLocalMon = NULL;
LPWSTR pPorts = NULL;
if( !pMonitorInit->bLocal ){
return NULL;
}
do {
FreeSplMem((LPVOID)pPorts);
dwCharCount *= 2;
pPorts = (LPWSTR) AllocSplMem(dwCharCount*sizeof(WCHAR));
if ( !pPorts ) {
DBGMSG(DBG_ERROR,
("Failed to alloc %d characters for ports\n", dwCharCount));
goto Fail;
}
rc = GetProfileString(szPorts, NULL, szNULL, pPorts, dwCharCount);
if ( !rc || dwCharCount >= 1024*1024 ) { // Work around in GetProfile-
String bug
DBGMSG(DBG_ERROR,
("GetProfilesString failed with %d\n", GetLastError()));
goto Fail;
}
} while ( rc >= dwCharCount - 2 );
pIniLocalMon = (PINILOCALMON)AllocSplMem( sizeof( INILOCALMON ));
```

```
if( !pIniLocalMon ){
goto Fail;
}
pIniLocalMon->signature = ILM_SIGNATURE;
pIniLocalMon->pMonitorInit = pMonitorInit;
//
// dwCharCount is now the count of return buffer, not including
// the NULL terminator. When we are past pPorts[rc], then
// we have parsed the entire string.
//
dwCharCount = rc;
LcmEnterSplSem();
//
// We now have all the ports
//
for( j = 0; j <= dwCharCount; j += rc + 1 ){
pPortTmp = pPorts + j;
rc = wcslen(pPortTmp);
if( !rc ){
continue;
}
if (!_wcsnicmp(pPortTmp, L"Ne", 2)) {
i = 2;
//
// For Ne-ports
//
if ( rc > 2 && pPortTmp[2] == L'-' )
++i;
for ( ; i < rc - 1 && iswdigit(pPortTmp[i]) ; ++i )
;
if ( i == rc - 1 && pPortTmp[rc-1] == L':' ) {
continue;
}
}
```

```
LcmCreatePortEntry(pIniLocalMon, pPortTmp);
}
FreeSplMem(pPorts);
LcmLeaveSplSem();
CheckAndAddIrdaPort(pIniLocalMon);
*phMonitor = (HANDLE)pIniLocalMon;
return &Monitor2;
Fail:
FreeSplMem( pPorts );
FreeSplMem( pIniLocalMon );
return NULL;
}
/*
*
*/
BOOL
DllEntryPoint(
HANDLE hModule,
DWORD dwReason,
LPVOID lpRes)
{
switch (dwReason)
{
case DLL_PROCESS_ATTACH:
LocalMonInit(hModule);
DisableThreadLibraryCalls(hModule);
return TRUE;
case DLL_PROCESS_DETACH:
return TRUE;
}
UNREFERENCED_PARAMETER(lpRes);
return TRUE;
}
```

П2.2. Исходные коды фильтр-драйвера

Здесь приводится листинг главного файла в фильтр-драйвере, архитектура и принципы написания которого рассматривались на страницах книги.

Листинг П2.2. Файл filter.c

```
/*++
Copyright (c) 1996 Microsoft Corporation
Module Name:
filter.c
Abstract: NULL filter driver -- boilerplate code
Author:
ervinp
Environment:
Kernel mode
Revision History:
--*/
#include <WDM.H>
#include "filter.h"
#ifdef ALLOC_PRAGMA
#pragma alloc_text(INIT, DriverEntry)
#pragma alloc_text(PAGE, VA_AddDevice)
#pragma alloc_text(PAGE, VA_DriverUnload)
#endif
NTSTATUS DriverEntry(
IN PDRIVER_OBJECT DriverObject,
IN PUNICODE_STRING RegistryPath
)
/*++
Routine Description:
Installable driver initialization entry point.
This entry point is called directly by the I/O system.
Arguments:
DriverObject - pointer to the driver object
RegistryPath - pointer to a unicode string representing the path,
```

```
to driver-specific key in the registry.
Return Value:
STATUS_SUCCESS if successful,
STATUS_UNSUCCESSFUL otherwise
--*/
{
ULONG i;
PAGED_CODE();
UNREFERENCED_PARAMETER(RegistryPath);
DBGOUT(("DriverEntry"));
/*
* Route all IRPs on device objects created by this driver
* to our IRP dispatch routine.
*/
for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++){
DriverObject->MajorFunction[i] = VA_Dispatch;
}
DriverObject->DriverExtension->AddDevice = VA_AddDevice;
DriverObject->DriverUnload = VA_DriverUnload;
return STATUS_SUCCESS;
}
NTSTATUS VA_AddDevice(
IN PDRIVER_OBJECT driverObj,
IN PDEVICE_OBJECT physicalDevObj
)
/*++
Routine Description:
The PlugPlay subsystem is handing us a brand new
PDO (Physical Device Object), for which we
(by means of INF registration) have been asked to filter.
We need to determine if we should attach or not.
Create a filter device object to attach to the stack
Initialize that device object
Return status success.
Remember: we can NOT actually send ANY non pnp IRPS to the given driver
```

```
stack, UNTIL we have received an IRP_MN_START_DEVICE.
Arguments:
driverObj - pointer to a device object.
physicalDevObj - pointer to a physical device object pointer
created by the underlying bus driver.
Return Value:
NT status code.
--*/
{
NTSTATUS status;
PDEVICE_OBJECT filterDevObj = NULL;
PAGED_CODE();
DBGOUT(("VA_AddDevice: drvObj=%ph, pdo=%ph", driverObj, physicalDevObj));
status = IoCreateDevice( driverObj,
sizeof(struct DEVICE_EXTENSION),
NULL, // name for this device
FILE_DEVICE_UNKNOWN,
FILE_AUTOGENERATED_DEVICE_NAME, // device characteristics
FALSE, // not exclusive
&filterDevObj); // our device object
if (NT_SUCCESS(status)){
struct DEVICE_EXTENSION *devExt;
ASSERT(filterDevObj);
/*
* Initialize device extension for new device object
*/
devExt = (struct DEVICE_EXTENSION *)filterDevObj->DeviceExtension;
RtlZeroMemory(devExt, sizeof(struct DEVICE_EXTENSION));
devExt->signature = DEVICE_EXTENSION_SIGNATURE;
devExt->state = STATE_INITIALIZED;
devExt->filterDevObj = filterDevObj;
devExt->physicalDevObj = physicalDevObj;
devExt->pendingActionCount = 0;
KeInitializeEvent(&devExt->removeEvent, NotificationEvent, FALSE);
#ifdef HANDLE_DEVICE_USAGE
```

```
KeInitializeEvent(&devExt->deviceUsageNotificationEvent, Synchronizatio-
nEvent, TRUE);
#endif // HANDLE_DEVICE_USAGE
/*
* Attach the new device object to the top of the device stack.
*/
devExt->topDevObj = IoAttachDeviceToDeviceStack(filterDevObj, physicalDe-
vObj);
ASSERT(devExt->topDevObj);
DBGOUT(("created filterDevObj %ph attached to %ph.", filterDevObj, de-
vExt->topDevObj));
//
// As a filter driver, we do not want to change the power or I/O
// behavior of the driver stack in any way. Recall that a filter
// driver should "appear" the same (almost) as the underlying device.
// Therefore we must copy some bits from the device object _directly_
// below us in the device stack (notice: DON'T copy from the PDO!)
//
/* Various I/O-related flags which should be maintained */
/* (copy from lower device object) */
filterDevObj->Flags |=
(devExt->topDevObj->Flags & (DO_BUFFERED_IO | DO_DIRECT_IO));
/* Various Power-related flags which should be maintained */
/* (copy from lower device object) */
filterDevObj->Flags |= (devExt->topDevObj->Flags &
(DO_POWER_INRUSH | DO_POWER_PAGABLE /*| DO_POWER_NOOP*/));
#ifdef HANDLE_DEVICE_USAGE
//
// To determine whether some of our routines should initially be
// pageable, we must consider the DO_POWER_xxxx flags of the
// device object directly below us in the device stack.
//
// * We make ourselves pageable if:
// - that devobj has its PAGABLE bit set (so we know our power
// routines won't be called at DISPATCH_LEVEL)
// -OR-
```

```
// - that devobj has its NOOP bit set (so we know we won't be
// participating in power-management at all). NOTE, currently
// DO_POWER_NOOP is not implemented.
//
// * Otherwise, we make ourselves non-pageable because either:
// - that devobj has its INRUSH bit set (so we also have to be
// INRUSH, and code that handles INRUSH irps can't be pageable)
// -OR-
// - that devobj does NOT have its PAGABLE bit set (and NOOP isn't
// set, so some of our code might be called at DISPATCH_LEVEL)
//
if ((devExt->topDevObj->Flags & DO_POWER_PAGABLE)
/*|| (devExt->topDevObj->Flags & DO_POWER_NOOP)*/)
{
// We're initially pageable.
//
// Don't need to do anything else here, for now.
}
else
{
// We're initially non-pageable.
//
// We need to lock-down the code for all routines
// that could be called at IRQL >= DISPATCH_LEVEL.
DBGOUT(( "LOCKing some driver code (non-pageable) (b/c init conditions)"
));
devExt->initUnlockHandle = MmLockPagableCodeSection( VA_Power ); // some
func that's inside the code section that we want to lock
ASSERT( NULL != devExt->initUnlockHandle );
}
/*
* Remember our initial flag settings.
* (Need remember initial settings to correctly handle
* setting of PAGABLE bit later.)
*/
devExt->initialFlags = filterDevObj->Flags & ~DO_DEVICE_INITIALIZING;
```

```
#endif // HANDLE_DEVICE_USAGE
/*
* Clear the initializing bit from the new device object's flags.
* NOTE: must not do this until *after* setting DO_POWER_xxxx flags
*/
filterDevObj->Flags &= ~DO_DEVICE_INITIALIZING;
/*
* This is a do-nothing call to a sample function which
* demonstrates how to read the device's registry area.
* Note that you cannot make this call on devExt->filterDevObj
* because a filter device object does not have a devNode.
* We pass devExt->physicalDevObj, which is the device object
* for which this driver is a filter driver.
*/
RegistryAccessSample(devExt, devExt->physicalDevObj);
}
ASSERT(NT_SUCCESS(status));
return status;
}
VOID VA_DriverUnload(IN PDRIVER_OBJECT DriverObject)
/*++
Routine Description:
Free all the allocated resources, etc.
Note: Although the DriverUnload function often does nothing,
the driver must set a DriverUnload function in
DriverEntry; otherwise, the kernel will never unload
the driver.
Arguments:
DriverObject - pointer to a driver object.
Return Value:
VOID.
--*/
{
PAGED_CODE();
DBGOUT(("VA_DriverUnload"));
}
```

```
NTSTATUS VA_Dispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
/*++
Routine Description:
Common entrypoint for all Io Request Packets
Arguments:
DeviceObject - pointer to a device object.
Irp - Io Request Packet
Return Value:
NT status code.
--*/
{
struct DEVICE_EXTENSION *devExt;
PIO_STACK_LOCATION irpSp;
BOOLEAN passIrpDown = TRUE;
UCHAR majorFunc, minorFunc;
NTSTATUS status;
devExt = DeviceObject->DeviceExtension;
ASSERT(devExt->signature == DEVICE_EXTENSION_SIGNATURE);
irpSp = IoGetCurrentIrpStackLocation(Irp);
/*
* Get major/minor function codes in private variables
* so we can access them after the IRP is completed.
*/
majorFunc = irpSp->MajorFunction;
minorFunc = irpSp->MinorFunction;
DBGOUT(("VA_Dispatch: majorFunc=%d, minorFunc=%d",
(ULONG)majorFunc, (ULONG)minorFunc));
/*
* For all IRPs except REMOVE, we increment the PendingActionCount
* across the dispatch routine in order to prevent a race condition with
* the REMOVE_DEVICE IRP (without this increment, if REMOVE_DEVICE
* preempted another IRP, device object and extension might get
* freed while the second thread was still using it).
*/
if (!((majorFunc == IRP_MJ_PNP) && (minorFunc == IRP_MN_REMOVE_DEVICE))){
```

```
IncrementPendingActionCount(devExt);
}
if ((majorFunc != IRP_MJ_PNP) &&
(majorFunc != IRP_MJ_CLOSE) &&
((devExt->state == STATE_REMOVING) ||
(devExt->state == STATE_REMOVED))){
/*
* While the device is being removed,
* we only pass down the PNP and CLOSE IRPs.
* We fail all other IRPs.
*/
status = Irp->IoStatus.Status = STATUS_DELETE_PENDING;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
passIrpDown = FALSE;
}
else {
switch (majorFunc){
case IRP_MJ_PNP:
status = VA_PnP(devExt, Irp);
passIrpDown = FALSE;
break;
case IRP_MJ_POWER:
status = VA_Power(devExt, Irp);
passIrpDown = FALSE;
break;
case IRP_MJ_CREATE:
case IRP_MJ_CLOSE:
case IRP_MJ_DEVICE_CONTROL:
case IRP_MJ_SYSTEM_CONTROL:
case IRP_MJ_INTERNAL_DEVICE_CONTROL:
default:
/*
* For unsupported IRPs, we simply send the IRP
* down the driver stack.
*/
```

```
break;
}
}
if (passIrpDown){
IoCopyCurrentIrpStackLocationToNext(Irp);
status = IoCallDriver(devExt->topDevObj, Irp);
}
/*
* Balance the increment to PendingActionCount above.
*/
if (!((majorFunc == IRP_MJ_PNP) && (minorFunc == IRP_MN_REMOVE_DEVICE))){
DecrementPendingActionCount(devExt);
}
return status;
}
```

Файл filter.h

```
/*++
Copyright (c) 1996 Microsoft Corporation
Module Name:
filter.h
Abstract: NULL filter driver -- boilerplate code
Author:
ervinp
Environment:
Kernel mode
Revision History:
--*/
//
// If this driver is going to be a filter in the paging, hibernation, or
dump
// file path, then HANDLE_DEVICE_USAGE should be defined.
//
// #define HANDLE_DEVICE_USAGE 1
enum deviceState {
STATE_INITIALIZED,
```

```
STATE_STARTING,
STATE_STARTED,
STATE_START_FAILED,
STATE_STOPPED, // implies device was previously started successfully
STATE_SUSPENDED,
STATE_REMOVING,
STATE_REMOVED
};
/*
* Memory tag for memory blocks allocated by this driver
* (used in ExAllocatePoolWithTag() call).
* This DWORD appears as "Filt" in a little-endian memory byte dump.
*
* NOTE: PLEASE change this value to be unique for your driver! Otherwise,
* your allocations will show up with every other driver that uses 'tliF'
as
* an allocation tag.
*
*/
#define FILTER_TAG (ULONG)'tliF'
#undef ExAllocatePool
#define ExAllocatePool(type, size) \
ExAllocatePoolWithTag (type, size, FILTER_TAG)
#define DEVICE_EXTENSION_SIGNATURE 'rtlF'
typedef struct DEVICE_EXTENSION {
/*
* Memory signature of a device extension, for debugging.
*/
ULONG signature;
/*
* Plug-and-play state of this device object.
*/
enum deviceState state;
/*
* The device object that this filter driver created.
```

```
*/
PDEVICE_OBJECT filterDevObj;
/*
* The device object created by the next lower driver.
*/
PDEVICE_OBJECT physicalDevObj;
/*
* The device object at the top of the stack that we attached to.
* This is often (but not always) the same as physicalDevObj.
*/
PDEVICE_OBJECT topDevObj;
/*
* deviceCapabilities includes a
* table mapping system power states to device power states.
*/
DEVICE_CAPABILITIES deviceCapabilities;
/*
* pendingActionCount is used to keep track of outstanding actions.
* removeEvent is used to wait until all pending actions are
* completed before complete the REMOVE_DEVICE IRP and let the
* driver get unloaded.
*/
LONG pendingActionCount;
KEVENT removeEvent;
#ifdef HANDLE_DEVICE_USAGE
/*
* Keep track of the number of paging/hibernation/crashdump
* files that are opened on this device.
*/
ULONG pagingFileCount, hibernationFileCount, crashdumpFileCount;
KEVENT deviceUsageNotificationEvent;
PVOID pagingPathUnlockHandle; /* handle to lock certain code as non-
pageable */
/*
* Also, might need to lock certain driver code as non-pageable, based on
```

```
* initial conditions (as opposed to paging-file considerations).
*/
PVOID initUnlockHandle;
ULONG initialFlags;
#endif // HANDLE_DEVICE_USAGE
};
#if DBG
#define DBGOUT(params_in_parentheses) \
{ \
DbgPrint("'FILTER> "); \
DbgPrint params_in_parentheses; \
DbgPrint("\n"); \
}
#define TRAP(msg) \
{ \
DBGOUT(("TRAP at file %s, line %d: '%s'.", __FILE__, __LINE__, msg)); \
DbgBreakPoint(); \
}
#else
#define DBGOUT(params_in_parentheses)
#define TRAP(msg)
#endif
/*
* Function externs
*/
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath);
NTSTATUS VA_AddDevice(IN PDRIVER_OBJECT driverObj, IN PDEVICE_OBJECT
pdo);
VOID VA_DriverUnload(IN PDRIVER_OBJECT DriverObject);
NTSTATUS VA_Dispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS VA_PnP(struct DEVICE_EXTENSION *devExt, PIRP irp);
#ifdef HANDLE_DEVICE_USAGE
NTSTATUS VA_DeviceUsageNotification(struct DEVICE_EXTENSION *devExt, PIRP
irp);
```

```
#endif // HANDLE_DEVICE_USAGE

NTSTATUS VA_Power(struct DEVICE_EXTENSION *devExt, PIRP irp);

NTSTATUS VA_PowerComplete(IN PDEVICE_OBJECT devObj, IN PIRP irp, IN PVOID
context);

NTSTATUS GetDeviceCapabilities(struct DEVICE_EXTENSION *devExt);

NTSTATUS CallNextDriverSync(struct DEVICE_EXTENSION *devExt, PIRP irp);

NTSTATUS CallDriverSync(PDEVICE_OBJECT devObj, PIRP irp);

NTSTATUS CallDriverSyncCompletion(IN PDEVICE_OBJECT devObj, IN PIRP irp,
IN PVOID Context);

VOID IncrementPendingActionCount(struct DEVICE_EXTENSION *devExt);

VOID DecrementPendingActionCount(struct DEVICE_EXTENSION *devExt);

NTSTATUS QueryDeviceKey(HANDLE Handle, PWCHAR ValueNameString, PVOID Da-
ta, ULONG DataLength);

VOID RegistryAccessSample(struct DEVICE_EXTENSION *devExt, PDEVICE_OBJECT
devObj);
```

Приложение 3

Полезные исходные коды из KMDF

Здесь приводятся полезные исходные коды из Windows KMDF. Используемая версия продукта — 1.0.

Текст файла приводится без каких-либо изменений, "купюр" и с полным сохранением указаний авторских прав.

Здесь приводится листинг главного файла в простейшем KMDF-драйвере, написание которого рассматривалось на страницах этой книги.

Листинг П3.1. Файл toaster.c — простейший KMDF-драйвер

```
/*++

Copyright (c) Microsoft Corporation.  All rights reserved.

  THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
  KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR
  PURPOSE.

Module Name:

  Toaster.c

Abstract:

  This is a simple form of function driver for toaster device. The driver
  doesn't handle any PnP and Power events because the framework provides
```

```
  default behavior for those events. This driver has enough support to
  allow an user application (toast/notify.exe) to open the device
  interface registered by the driver and send read, write or ioctl
  requests.

Environment:

  Kernel mode


Revision History:

  Eliyas Yakub - 10-Oct-2002

  Framework Team Jan 22 2003 - Converted to use the Driver Framework.

--*/
#include "toaster.h"

#ifdef ALLOC_PRAGMA
#pragma alloc_text (INIT, DriverEntry)
#pragma alloc_text (PAGE, ToasterEvtDeviceAdd)
#pragma alloc_text (PAGE, ToasterEvtIoRead)
#pragma alloc_text (PAGE, ToasterEvtIoWrite)
#pragma alloc_text (PAGE, ToasterEvtIoDeviceControl)
#endif


NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT  DriverObject,
  IN PUNICODE_STRING RegistryPath
  )
/*++
```

```
Routine Description:
  DriverEntry initializes the driver and is the first routine called
  by the system after the driver is loaded. DriverEntry configures and
  creates a WDF driver object.

Parameters Description:

  DriverObject - represents the instance of the function driver that is
  loaded into memory. DriverObject is allocated by the system before the
  driver is loaded, and it is released by the system after the system
  unloads the function driver from memory.

  RegistryPath - represents the driver specific path in the Registry.
  The function driver can use the path to store driver related data
  between reboots. The path does not store hardware instance specific
  data.

Return Value:

  STATUS_SUCCESS if successful,
  STATUS_UNSUCCESSFUL otherwise.

--*/
{
  NTSTATUS      status = STATUS_SUCCESS;
  WDF_DRIVER_CONFIG   config;

  KdPrint(("Toaster Function Driver Sample - Driver Framework Edi-
tion.\n"));
  KdPrint(("Built %s %s\n", __DATE__, __TIME__));

  //
  // Initialize driver config to control the attributes that
  // are global to the driver. Note that framework by default
  // provides a driver unload routine. If DriverEntry creates any re-
sources
```

```
  // that require clean-up in driver unload,
  // you can manually override the default by supplying a pointer to the
EvtDriverUnload
  // callback in the config structure. In general xxx_CONFIG_INIT macros
are provided to
  // initialize most commonly used members.
  //

  WDF_DRIVER_CONFIG_INIT(
    &config,
    ToasterEvtDeviceAdd
    );



  //
  // Create a framework driver object to represent our driver.
  //
  status = WdfDriverCreate(
    DriverObject,
    RegistryPath,
    WDF_NO_OBJECT_ATTRIBUTES, // Driver Attributes
    &config,      // Driver Config Info
    WDF_NO_HANDLE
    );

  if (!NT_SUCCESS(status)) {
    KdPrint( ("WdfDriverCreate failed with status 0x%x\n", status));
  }

  return status;
}



NTSTATUS
ToasterEvtDeviceAdd(
  IN WDFDRIVER     Driver,
```

```
  IN PWDFDEVICE_INIT DeviceInit
  )
/*++
Routine Description:

  ToasterEvtDeviceAdd is called by the framework in response to AddDevice
  call from the PnP manager. We create and initialize a WDF device object
  to represent a new instance of toaster device.

Arguments:

  Driver - Handle to a framework driver object created in DriverEntry

  DeviceInit - Pointer to a framework-allocated WDFDEVICE_INIT structure.

Return Value:

  NTSTATUS

--*/
{
  NTSTATUS          status = STATUS_SUCCESS;
  PFDO_DATA         fdoData;
  WDF_IO_QUEUE_CONFIG  queueConfig;
  WDF_OBJECT_ATTRIBUTES  fdoAttributes;
  WDFDEVICE         hDevice;
  WDFQUEUE          queue;

  UNREFERENCED_PARAMETER(Driver);

  PAGED_CODE();

  KdPrint(("ToasterEvtDeviceAdd called\n"));

  //
  // Initialize attributes and a context area for the device object.
```

```
//
//
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&fdoAttributes, FDO_DATA);

//
// Create a framework device object.This call will in turn create
// a WDM device object, attach to the lower stack, and set the
// appropriate flags and attributes.
//
status = WdfDeviceCreate(&DeviceInit, &fdoAttributes, &hDevice);
if (!NT_SUCCESS(status)) {
  KdPrint( ("WdfDeviceCreate failed with status code 0x%x\n", status));
  return status;
}

//
// Get the device context by using the accessor function specified in
// the WDF_DECLARE_CONTEXT_TYPE_WITH_NAME macro for FDO_DATA.
//
fdoData = ToasterFdoGetData(hDevice);

//
// Tell the Framework that this device will need an interface
//
status = WdfDeviceCreateDeviceInterface(
       hDevice,
       (LPGUID) &GUID_DEVINTERFACE_TOASTER,
       NULL // ReferenceString
     );

if (!NT_SUCCESS (status)) {
  KdPrint( ("WdfDeviceCreateDeviceInterface failed 0x%x\n", status));
  return status;
}
```

```
//
// Register I/O callbacks to tell the framework that you are interested
// in handling IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MJ_DEVICE_CONTROL
// requests.
// If a specific callback function is not specified for one ofthese,
// the request will be dispatched to the EvtIoDefault handler, if any.
// If there is no EvtIoDefault handler, the request will be failed with
// STATUS_INVALID_DEVICE_REQUEST.
// WdfIoQueueDispatchParallel means that we are capable of handling
// all the I/O requests simultaneously and we are responsible for
// protecting
// data that could be accessed by these callbacks simultaneously.
// A default queue gets all the requests that are not
// configured for forwarding using
// WdfDeviceConfigureRequestDispatching.
//
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&queueConfig,
                        WdfIoQueueDispatchParallel);

queueConfig.EvtIoRead = ToasterEvtIoRead;
queueConfig.EvtIoWrite = ToasterEvtIoWrite;
queueConfig.EvtIoDeviceControl = ToasterEvtIoDeviceControl;

status = WdfIoQueueCreate(
  hDevice,
  &queueConfig,
  WDF_NO_OBJECT_ATTRIBUTES,
  &queue
  );

if (!NT_SUCCESS (status)) {

  KdPrint( ("WdfIoQueueCreate failed 0x%x\n", status));
  return status;
}
```

```
  return status;
}

VOID
ToasterEvtIoRead (
  WDFQUEUE    Queue,
  WDFREQUEST  Request,
  size_t    Length
  )
/*++

Routine Description:

  Performs read from the toaster device. This event is called when the
  framework receives IRP_MJ_READ requests.

Arguments:

  Queue -  Handle to the framework queue object that is associated with
the
       I/O request.
  Request - Handle to a framework request object.

  Lenght - Length of the data buffer associated with the request.
         By default, the queue does not dispatch
         zero length read & write requests to the driver and instead to
         complete such requests with status success. So we will never get
         a zero length request.

Return Value:

  None.

--*/
{
  NTSTATUS  status;
  ULONG_PTR bytesCopied =0;
  WDFMEMORY memory;
```

```
  UNREFERENCED_PARAMETER(Queue);
  UNREFERENCED_PARAMETER(Length);

  PAGED_CODE();

  KdPrint(( "ToasterEvtIoRead: Request: 0x%p, Queue: 0x%p\n",
                Request, Queue));

  //
  // Get the request memory and perform read operation here
  //
  status = WdfRequestRetrieveOutputMemory(Request, &memory);
  if(NT_SUCCESS(status) ) {
    //
    // Copy data into the memory buffer using WdfMemoryCopyFromBuffer
    //
  }

  WdfRequestCompleteWithInformation(Request, status, bytesCopied);
}

VOID
ToasterEvtIoWrite (
  WDFQUEUE    Queue,
  WDFREQUEST  Request,
  size_t    Length
  )
/*++

Routine Description:

  Performs write to the toaster device. This event is called when the
  framework receives IRP_MJ_WRITE requests.

Arguments:
```

```
  Queue -  Handle to the framework queue object that is associated with
the
      I/O request.
  Request - Handle to a framework request object.

  Lenght - Length of the data buffer associated with the request.
         The default property of the queue is to not dispatch
         zero lenght read & write requests to the driver and
         complete is with status success. So we will never get
         a zero length request.

Return Value:

   None
--*/

{
  NTSTATUS  status;
  ULONG_PTR bytesWritten =0;
  WDFMEMORY memory;

  UNREFERENCED_PARAMETER(Queue);
  UNREFERENCED_PARAMETER(Length);

  KdPrint(("ToasterEvtIoWrite. Request: 0x%p, Queue: 0x%p\n",
               Request, Queue));

  PAGED_CODE();

  //
  // Get the request buffer and perform write operation here
  //
  status = WdfRequestRetrieveInputMemory(Request, &memory);
  if(NT_SUCCESS(status) ) {
    //
    // 1) Use WdfMemoryCopyToBuffer to copy data from the request
```

```
    // to driver buffer.
    // 2) Or get the buffer pointer from the request by calling
    // WdfRequestRetrieveInputBuffer
    // 3) Or you can get the buffer pointer from the memory handle
    // by calling WdfMemoryGetBuffer.
    //
    bytesWritten = Length;
  }

  WdfRequestCompleteWithInformation(Request, status, bytesWritten);

}

VOID
ToasterEvtIoDeviceControl(
  IN WDFQUEUE   Queue,
  IN WDFREQUEST   Request,
  IN size_t     OutputBufferLength,
  IN size_t     InputBufferLength,
  IN ULONG    IoControlCode
  )
/*++
Routine Description:

  This event is called when the framework receives IRP_MJ_DEVICE_CONTROL
  requests from the system.

Arguments:

  Queue - Handle to the framework queue object that is associated
      with the I/O request.
  Request - Handle to a framework request object.

  OutputBufferLength - length of the request's output buffer,
           if an output buffer is available.
  InputBufferLength - length of the request's input buffer,
           if an input buffer is available.
```

```
  IoControlCode - the driver-defined or system-defined I/O control code
          (IOCTL) that is associated with the request.

Return Value:

   VOID

--*/
{
  NTSTATUS  status= STATUS_SUCCESS;

  UNREFERENCED_PARAMETER(Queue);
  UNREFERENCED_PARAMETER(OutputBufferLength);
  UNREFERENCED_PARAMETER(InputBufferLength);

  KdPrint(("ToasterEvtIoDeviceControl called\n"));

  PAGED_CODE();

  //
  // Use WdfRequestRetrieveInputBuffer and WdfRequestRetrieveOutputBuffer
  // to get the request buffers.
  //

  switch (IoControlCode) {

  default:
    status = STATUS_INVALID_DEVICE_REQUEST;
  }

  //
  // Complete the Request.
  //
  WdfRequestCompleteWithInformation(Request, status, (ULONG_PTR) 0);
}
```

Приложение 4

Полезные исходные коды из UMDF

Здесь приводятся полезные исходные коды из Windows UMDF. Используемая версия продукта — 1.0.

Тексты файлов приводятся без каких-либо изменений, "купюр" и с полным сохранением указаний авторских прав.

Далее приводятся листинги главных файлов в простейшем UMDF-драйвере, написание которого рассматривалось на страницах этой книги.

Листинг П4.1. Файл Device.c

```
/*++

Copyright (C) Microsoft Corporation, All Rights Reserved.

Module Name:

  Device.cpp

Abstract:

  This module contains the implementation of the UMDF Skeleton sample
  driver's device callback object.

  The skeleton sample device does very little.  It does not implement
  either of the PNP interfaces so once the device is setup, it won't
  ever get any callbacks until the device is removed.
```

```
Environment:

   Windows User-Mode Driver Framework (WUDF)

--*/

#include "internal.h"
#include "device.tmh"

HRESULT
CMyDevice::CreateInstance(
  __in IWDFDriver *FxDriver,
  __in IWDFDeviceInitialize * FxDeviceInit,
  __out PCMyDevice *Device
  )
/*++

  Routine Description:

  This method creates and initializs an instance of the skeleton driver's
  device callback object.

  Arguments:

  FxDeviceInit - the settings for the device.

  Device - a location to store the referenced pointer to the device
  object.

  Return Value:

  Status

--*/
{
  PCMyDevice device;
  HRESULT hr;
```

```
  //
  // Allocate a new instance of the device class.
  //

  device = new CMyDevice();

  if (NULL == device)
  {
    return E_OUTOFMEMORY;
  }

  //
  // Initialize the instance.
  //

  hr = device->Initialize(FxDriver, FxDeviceInit);

  if (SUCCEEDED(hr))
  {
    *Device = device;
  }
  else
  {
    device->Release();
  }

  return hr;
}

HRESULT
CMyDevice::Initialize(
  __in IWDFDriver       * FxDriver,
  __in IWDFDeviceInitialize * FxDeviceInit
  )
/*++
```

```
  Routine Description:

  This method initializes the device callback object and creates the
  partner device object.

  The method should perform any device-specific configuration that:
    *  could fail (these can't be done in the constructor)
    *  must be done before the partner object is created -or-
    *  can be done after the partner object is created and which aren't
       influenced by any device-level parameters the parent (the driver
       in this case) might set.

  Arguments:

  FxDeviceInit - the settings for this device.

  Return Value:

  status.

--*/
{
  IWDFDevice *fxDevice;
  HRESULT hr;

  //
  // Configure things like the locking model before we go to create our
  // partner device.
  //

  //
  // TODO: Set the locking model.  The skeleton uses device level
  //     locking, but you can choose "none" as well.
  //

  FxDeviceInit->SetLockingConstraint(WdfDeviceLevel);
```

```
//
// TODO: If you're writing a filter driver then indicate that here.
//
// FxDeviceInit->SetFilter();
//

//
// TODO: Any per-device initialization which must be done before
//     creating the partner object.
//

//
// Create a new FX device object and assign the new callback object to
// handle any device level events that occur.
//

//
// QueryIUnknown references the IUnknown interface that it returns
// (which is the same as referencing the device).  We pass that to
// CreateDevice, which takes its own reference if everything works.
//

{
  IUnknown *unknown = this->QueryIUnknown();

  hr = FxDriver->CreateDevice(FxDeviceInit, unknown, &fxDevice);

  unknown->Release();
}

//
// If that succeeded then set our FxDevice member variable.
//

if (SUCCEEDED(hr))
{
  m_FxDevice = fxDevice;
```

```
    //
    // Drop the reference we got from CreateDevice.  Since this object
    // is partnered with the framework object they have the same
    // lifespan - there is no need for an additional reference.
    //

    fxDevice->Release();
  }

  return hr;
}

HRESULT
CMyDevice::Configure(
  VOID
  )
/*++

  Routine Description:

  This method is called after the device callback object has been
  initialized and returned to the driver.  It would setup the device's
  queues and their corresponding callback objects.

  Arguments:

  FxDevice - the framework device object for which we're handling events.

  Return Value:

  status

--*/
{
  //
```

```
  // TODO: Setup your device queues and I/O forwarding.
  //

  return S_OK;
}

HRESULT
CMyDevice::QueryInterface(
  __in REFIID InterfaceId,
  __out PVOID *Object
  )
/*++

  Routine Description:

  This method is called to get a pointer to one of the object's callback
  interfaces.

  Since the skeleton driver doesn't support any of the device events,
  this method simply calls the base class's BaseQueryInterface.

  If the skeleton is extended to include device event interfaces then
  this method must be changed to check the IID and return pointers
  to them as appropriate.

  Arguments:

  InterfaceId - the interface being requested

  Object - a location to store the interface pointer if successful

  Return Value:

  S_OK or E_NOINTERFACE
```

```
--*/
{
  return CUnknown::QueryInterface(InterfaceId, Object);
}
```

Листинг П4.2. Файл Driver.c

```
/*++

Copyright (C) Microsoft Corporation, All Rights Reserved.

Module Name:

  Driver.cpp

Abstract:

  This module contains the implementation of the UMDF Skeleton Sample's
  core driver callback object.

Environment:

   Windows User-Mode Driver Framework (WUDF)

--*/

#include "internal.h"
#include "driver.tmh"

HRESULT
CMyDriver::CreateInstance(
  __out PCMyDriver *Driver
  )
/*++

  Routine Description:
```

```
  This static method is invoked in order to create and initialize a new
  instance of the driver class.  The caller should arrange for the object
  to be released when it is no longer in use.

  Arguments:

  Driver - a location to store a referenced pointer to the new instance

  Return Value:

  S_OK if successful, or error otherwise.

--*/
{
  PCMyDriver driver;
  HRESULT hr;

  //
  // Allocate the callback object.
  //

  driver = new CMyDriver();

  if (NULL == driver)
  {
    return E_OUTOFMEMORY;
  }

  //
  // Initialize the callback object.
  //

  hr = driver->Initialize();

  if (SUCCEEDED(hr))
```

```
  {
    //
    // Store a pointer to the new, initialized object in the output
    // parameter.
    //

    *Driver = driver;
  }
  else
  {

    //
    // Release the reference on the driver object to get it to delete
    // itself.
    //

    driver->Release();
  }

  return hr;
}

HRESULT
CMyDriver::Initialize(
  VOID
  )
/*++

  Routine Description:

  This method is called to initialize a newly created driver callback
  object
  before it is returned to the creator.  Unlike the constructor, the
  Initialize method contains operations which could potentially fail.
```

```
  Arguments:

  None

  Return Value:

  None

--*/
{
  return S_OK;
}

HRESULT
CMyDriver::QueryInterface(
  __in REFIID InterfaceId,
  __out PVOID *Interface
  )
/*++

  Routine Description:

  This method returns a pointer to the requested interface
  on the callback object..

  Arguments:

  InterfaceId - the IID of the interface to query/reference

  Interface - a location to store the interface pointer.

  Return Value:

  S_OK if the interface is supported.
  E_NOINTERFACE if it is not supported.
```

```
--*/
{
  if (IsEqualIID(InterfaceId, __uuidof(IDriverEntry)))
  {
    *Interface = QueryIDriverEntry();
    return S_OK;
  }
  else
  {
    return CUnknown::QueryInterface(InterfaceId, Interface);
  }
}

HRESULT
CMyDriver::OnDeviceAdd(
  __in IWDFDriver *FxWdfDriver,
  __in IWDFDeviceInitialize *FxDeviceInit
  )
/*++

  Routine Description:

  The FX invokes this method when it wants to install our driver on
  a device stack. This method creates a device callback object, then
  calls the Fx to create an Fx device object and associate the new
  callback object with it.

  Arguments:

  FxWdfDriver - the Fx driver object.

  FxDeviceInit - the initialization information for the device.

  Return Value:

  status
```

```
--*/
{
  HRESULT hr;

  PCMyDevice device = NULL;

  //
  // TODO: Do any per-device initialization (reading settings from the
  //     registry for example) that's necessary before creating your
  //     device callback object here.  Otherwise you can leave such
  //     initialization to the initialization of the device event
  //     handler.
  //

  //
  // Create a new instance of our device callback object
  //

  hr = CMyDevice::CreateInstance(FxWdfDriver, FxDeviceInit, &device);

  //
  // TODO: Change any per-device settings that the object exposes before
  //     calling Configure to let it complete its initialization.
  //

  //
  // If that succeeded then call the device's construct method.  This
  // allows the device to create any queues or other structures that it
  // needs now that the corresponding fx device object has been created.
  //

  if (SUCCEEDED(hr))
  {
    hr = device->Configure();
  }
```

```
  //
  // Release the reference on the device callback object now that it's
  // been associated with an fx device object.
  //

  if (NULL != device)
  {
    device->Release();
  }

  return hr;
}
```

Список полезной литературы

Что касается литературы, то ничего более или менее вразумительного именно по разработке драйверов на русском языке порекомендовать нельзя — количество литературы на российском рынке по драйверным технологиям ничтожно мало.

А в общем, можно порекомендовать следующее:

1. Walter Oney. Programming the Microsoft Driver Model. — M.: Microsoft Press, 1999.
2. Неббет Г. Справочник по базовым функциям API Windows NT/2000. — М.: Вильямс, 2002.
3. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. — СПб.: Питер, 2001.
4. Шрайбер С. Недокументированные возможности Windows 2000. — СПб.: Питер, 2002.

Также напоминаю о документации, прилагающейся к Microsoft DDK (которая доступна и в online-режиме на сайте Microsoft), KMDF и UMDF (также доступна в online-режиме).

Предметный указатель

П

С

Т

У

Ф