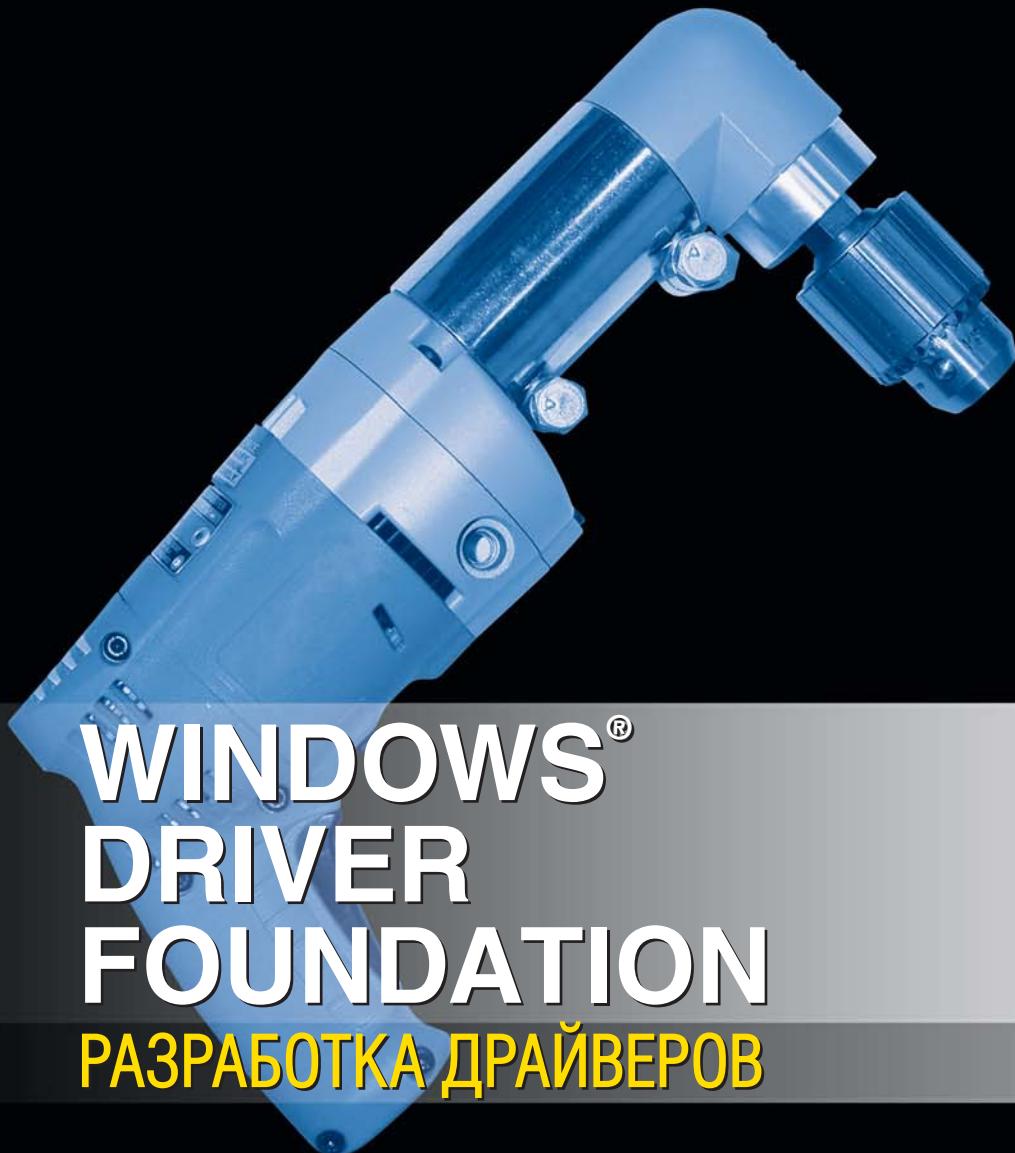


**Microsoft®**

Пенни Орвик  
Гай Смит



# WINDOWS® DRIVER FOUNDATION

## РАЗРАБОТКА ДРАЙВЕРОВ

**Microsoft®**

# Developing Drivers with the **Windows®** Driver Foundation

*Penny Orwick  
Guy Smith*

**Пенни Орвик  
Гай Смит**

**WINDOWS®  
DRIVER  
FOUNDATION  
РАЗРАБОТКА ДРАЙВЕРОВ**

«Русская Редакция»

2008

«БХВ-Петербург»

УДК 681.3.06  
ББК 32.973.26-018.2  
О-63

- Орвик, П.**
- O-63 Windows® Driver Foundation: разработка драйверов: Пер. с англ. / П. Орвик, Г. Смит. — М.: Издательство «Русская Редакция»; СПб.: «БХВ-Петербург», 2008. — 880 с.: ил.  
ISBN 978-5-7502-0364-2 («Русская Редакция»)  
ISBN 978-5-9775-0185-9 («БХВ-Петербург»)

Книга содержит описания принципов и методик, примеры программирования и подсказки для эффективной разработки драйверов. Представлены инструменты и ресурсы, основные понятия драйверов и операционной системы Windows, обзор модели Windows Driver Foundation (WDF), информация об архитектуре Windows, модели ввода/вывода. Приводятся три основные составляющие модели WDF: инфраструктура драйвера пользовательского режима, инфраструктура драйвера режима ядра и набор инструментов тестирования и верификации, а также шаблоны и рекомендации по выбору. Подробно рассматриваются различия между пользовательским режимом и режимом ядра.

*Для разработчиков аппаратного обеспечения и программистов,  
в том числе не имеющих опыта разработки драйверов*

УДК 681.3.06  
ББК 32.973.26-018.2

© 2007-2012, Translation Russian Edition Publishers.

Authorized Russian translation of the English edition of Developing Drivers with the Windows® Driver Foundation, ISBN 9780735623743 © Microsoft Corporation.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

© 2007-2012, перевод ООО «Издательство «Русская редакция», издательство «БХВ-Петербург».

Авторизованный перевод с английского на русский язык произведения Developing Drivers with the Windows® Driver Foundation, ISBN 9780735623743 © Microsoft Corporation.

Этот перевод оригинального издания публикуется и продается с разрешения O'Reilly Media, Inc., которая владеет или распоряжается всеми правами на его публикацию и продажу.

© 2008-2012, оформление и подготовка к изданию, ООО «Издательство «Русская редакция», издательство «БХВ-Петербург».

Microsoft, а также товарные знаки, перечисленные в списке, расположеннном по адресу:

<http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

# Оглавление

<b>Об авторах .....</b>	<b>1</b>
<b>Предисловие .....</b>	<b>3</b>
<b>Благодарности.....</b>	<b>7</b>
<b>ЧАСТЬ I. НАЧАЛО РАБОТЫ С МОДЕЛЬЮ WDF.....</b>	<b>11</b>
<b>Глава 1. Введение в WDF .....</b>	<b>13</b>
Об этой книге .....	14
Кому адресована эта книга .....	14
Часть I. Начало работы с моделью WDF .....	15
Часть II. Изучение инфраструктур .....	15
Часть III. Применение основ WDF .....	16
Часть IV. Смотрим глубже — больше о драйверах WDF .....	18
Часть V. Создание, установка и тестирование драйверов WDF.....	18
Словарь.....	19
Условные обозначения .....	19
Приступая к разработке драйверов.....	21
Системные требования для разработки драйверов.....	21
Как приобрести и установить набор WDK.....	22
Библиотеки WDK .....	24
Документация WDK .....	24
Инструменты WDK .....	25
Образцы WDK .....	26
Образцы UMDF .....	26
Образцы KMDF .....	27
Как приобрести проверочные версии Windows .....	27
Как приобрести отладочные инструменты.....	28
Как приобрести устройства обучения OSR .....	29
Основные источники информации .....	30
Основные ссылки.....	32
<b>Глава 2. Основы драйверов под Windows.....</b>	<b>33</b>
Что такое драйвер?.....	34
Базовая архитектура Windows.....	35

Архитектура драйвера.....	37
Объекты устройств и стек устройства .....	37
Дерево устройств.....	39
Объекты и структуры данных ядра.....	40
Модель ввода/вывода Windows.....	41
Запросы ввода/вывода.....	42
Обработка прерываний стеком устройства .....	42
Буферы данных и типы передачи ввода/вывода .....	43
Передача и получение данных от устройства .....	44
Plug and Play и управление энергопотреблением .....	45
Основы программирования в режиме ядра.....	45
Прерывания и уровни IRQL.....	46
Параллелизм и синхронизация .....	48
Потоки .....	48
Синхронизация .....	49
Состояние гонок и взаимоблокировки .....	50
Память .....	51
Управление памятью .....	51
Пулы памяти .....	52
Стек ядра.....	52
Списки MDL .....	53
Советы по программированию в режиме ядра.....	53
Выделение памяти .....	53
Использование спин-блокировок .....	54
Управление ошибками страниц .....	54
Обращение к памяти пользовательского режима .....	54
Блокирование потоков .....	55
Верификация драйверов .....	55
Использование макросов .....	55
Основные термины .....	56
<b>Глава 3. Основы WDF .....</b>	<b>60</b>
WDF и WDM .....	60
Что такое WDF? .....	61
Объектная модель WDF .....	62
Программный интерфейс.....	63
Иерархия объектов .....	63
Параллелизм и синхронизация .....	64
Модель ввода/вывода.....	64
Отмена запросов ввода/вывода .....	65
Исполнители ввода/вывода.....	65
Обработка некритических ошибок .....	66
Извещение об ошибках UMDF.....	67
Извещение об ошибках KMDF.....	67
Plug and Play и управление энергопотреблением .....	67
Критерии конструкции WDF для Plug and Play и управления энергопотреблением.....	68
Безопасность.....	69
Стандартные безопасные установки .....	70
Проверка достоверности параметров .....	70
Поддержка верификации, трассировки и отладки.....	70
Обслуживаемость и управление версиями.....	71

---

<b>ЧАСТЬ II. ИЗУЧЕНИЕ ИНФРАСТРУКТУР</b>	<b>73</b>
<b>Глава 4. Обзор драйверных инфраструктур</b>	<b>75</b>
Обзор драйверных инфраструктур .....	75
Обзор инфраструктуры UMDF .....	76
Объекты инфраструктуры UMDF .....	77
Объекты обратного вызова инфраструктуры UMDF .....	78
Обзор инфраструктуры KMDF .....	78
Объекты KMDF .....	79
Функции обратного вызова инфраструктуры KMDF .....	80
Архитектура WDF .....	80
Инфраструктура UMDF .....	83
Компоненты инфраструктуры UMDF.....	84
Критические ошибки в драйверах UMDF .....	86
Типичный запрос ввода/вывода UMDF .....	87
Инфраструктура KMDF .....	88
Компоненты инфраструктуры KMDF.....	88
Критические ошибки в драйверах KMDF .....	90
Типичный запрос ввода/вывода KMDF .....	90
Поддержка устройств и драйверов в WDF.....	91
Устройства, поддерживаемые UMDF .....	91
Достоинства драйверов UMDF .....	92
Недостатки драйверов UMDF .....	93
Устройства, поддерживаемые KMDF .....	94
Как выбрать правильную инфраструктуру .....	94
<b>Глава 5. Объектная модель WDF .....</b>	<b>96</b>
Обзор объектной модели .....	96
Методы, свойства и события .....	97
Обратные вызовы по событию .....	98
Атрибуты объектов .....	98
Иерархия и время жизни объектов .....	99
Контекст объекта.....	99
Реализация объектной модели UMDF .....	100
Соглашение об именах UMDF .....	100
Объекты и интерфейсы инфраструктуры UMDF.....	101
Объекты обратного вызова и интерфейсы драйвера UMDF .....	103
Пример UMDF: объекты и интерфейсы обратных вызовов .....	105
Реализация объектной модели KMDF .....	106
Типы объектов KMDF.....	107
Соглашение об именах KMDF .....	109
Создание объектов .....	110
Создание объектов UMDF .....	110
Создание объектов KMDF .....	111
Структура конфигурации объекта KMDF .....	111
Структура атрибутов объекта KMDF .....	112
Методы создания объектов KMDF .....	113
Иерархия и время жизни объектов .....	114
Удаление, ликвидация, очистка и уничтожение .....	115
Иерархия объектов UMDF.....	116
Иерархия объектов KMDF.....	117
Удаление объектов .....	118
Обратные вызовы очистки ресурсов.....	121

Обратные вызовы деструкции .....	121
Удаление объектов UMDF .....	122
Удаление объектов KMDF .....	122
Исключение: завершенные запросы ввода/вывода .....	124
Область контекста объекта .....	125
Данные контекста объекта UMDF .....	126
Информация контекста UMDF в членах данных объекта обратного вызова .....	127
Область контекста UMDF для объекта инфраструктуры .....	128
Область контекста объекта KMDF .....	129
Объявления типа области контекста KMDF .....	130
Инициализация полей контекста в структуре атрибутов объекта .....	130
Назначение области контекста KMDF объекту .....	131
<b>Глава 6. Структура драйвера и его инициализация .....</b>	<b>132</b>
Обязательные компоненты драйвера .....	132
Структура драйверов UMDF и требования к ним .....	133
Структура драйверов KMDF и требования к ним .....	136
Объект драйвера .....	138
Создание объекта обратного вызова для драйвера UMDF .....	138
Создание объекта драйвера KMDF .....	140
Объекты устройств .....	142
Типы объектов устройств .....	142
Драйверы фильтра и объекты устройств фильтра .....	143
Функциональные драйверы и объекты функциональных устройств .....	144
Драйверы шины и объекты физических устройств (KMDF) .....	144
Драйверы унаследованных устройств и объекты устройств управления (KMDF) .....	146
Драйверы WDF, типы драйверов и типы объектов устройств .....	147
Свойства устройств .....	147
Инициализирование объекта устройства .....	149
Объекты очередей и другие вспомогательные объекты .....	149
Интерфейсы устройств .....	150
Интерфейсы устройств и символьные ссылки .....	151
Создание и инициализирование объекта устройства UMDF .....	151
Создание объекта обратного вызова устройства .....	152
Создание и инициализирование инфраструктурного объекта устройства .....	152
Пример UMDF: интерфейс устройства .....	154
Создание и инициализирование объекта устройства KMDF .....	155
Структура инициализации устройства KMDF .....	155
Инициализация KMDF объектов FDO .....	156
Инициализация KMDF объектов FiDO .....	156
Область контекста объекта устройства .....	157
Создание объектов устройств KMDF .....	157
Дополнительные задачи, выполняемые функцией <i>EvtDriverDeviceAdd</i> .....	158
Пример KMDF: функция обратного вызова <i>EvtDriverDeviceAdd</i> .....	158
Перечисление дочерних устройств (только PDO-объекты KMDF) .....	160
Статическое и динамическое перечисление в драйверах шины .....	160
Динамическое перечисление .....	160
Статическое перечисление .....	161
Инициализация объектов PDO .....	161
Способы именования устройства для драйверов KMDF .....	162
Именованные объекты устройств .....	162
Дескрипторы безопасности .....	163

<b>ЧАСТЬ III. ПРИМЕНЕНИЕ ОСНОВ WDF.....</b>	<b>165</b>
<b>Глава 7. Plug and Play и управление энергопотреблением.....</b>	<b>167</b>
Введение в Plug and Play и управление энергопотреблением .....	168
О механизме Plug and Play.....	169
О состояниях энергопотребления .....	170
Политика энергопотребления.....	172
Plug and Play и управление энергопотреблением в WDF .....	173
Стандартные настройки Plug and Play и управления энергопотреблением.....	173
Очереди ввода/вывода и управление энергопотреблением .....	174
Обратные вызовы функций для событий Plug and Play и управления энергопотреблением .....	174
Поддержка бездействия и пробуждения (только для KMDF) .....	176
Энергостраничные и неэнергостраничные драйверы.....	177
Порядок обратных вызовов функций для событий Plug and Play и управление энергопотреблением .....	179
Перечисление и запуск устройств.....	183
Отключение питания и удаление устройства.....	186
Неожиданное извлечение.....	190
Последовательность действий при неожиданном удалении для драйверов UMDF .....	190
Последовательность действий при неожиданном удалении устройства для драйверов KMDF .....	191
Реализация Plug and Play и управления энергопотреблением в драйверах WDF .....	192
Чисто программные драйверы Plug and Play и управление энергопотреблением .....	193
Пример UMDF: чисто программный драйвер фильтра для Plug and Play.....	194
Пример KMDF: чисто программный драйвер фильтра для Plug and Play .....	195
Действия инфраструктуры для чисто программных драйверов.....	197
Plug and Play и управление энергопотреблением в простых аппаратных драйверах .....	197
Инициализация и deinициализация устройства при включении и выключении.....	198
Управление энергопотреблением очередей в аппаратных функциональных драйверах.....	199
Пример UMDF: код для Plug and Play и управления энергопотреблением в протокольном функциональном драйвере.....	200
Управляемая энергопотреблением очередь для драйвера UMDF .....	201
Методы интерфейса <i>IPnpCallbackHardware</i> .....	201
Методы интерфейса <i>IPnpCallback</i> .....	203
Пример KMDF: код для Plug and Play и управления энергопотреблением в простом аппаратном функциональном драйвере.....	204
Пример KMDF: регистрация обратных вызовов и организация управляемых энергопотреблением очередей .....	205
Пример KMDF: обратные вызовы для входа и выхода из состояния D0 .....	207
Действия инфраструктуры для простого аппаратного функционального драйвера.....	208
KMDF, устройства хранение и гибернация.....	210
Продвинутые возможности управления энергопотреблением для драйверов KMDF .....	210
Поддержка бездействия устройства в режиме низкого энергопотребления для драйверов KMDF .....	210
Настройки и управление бездействием в драйверах KMDF .....	211
Выбор периода простоя и состояний пониженного энергопотребления в драйверах KMDF .....	214
Поддержка пробуждения устройства для драйверов KMDF .....	215
Реализация пробуждения из состояния Sx в драйверах KMDF .....	217
Реализация пробуждения из состояния S0 в драйверах KMDF.....	219
Пример KMDF: поддержка бездействия и пробуждения устройств.....	220
Действия инфраструктуры для поддержки бездействия устройства .....	223
Действия инфраструктуры, поддерживающие пробуждение устройства.....	223

---

<b>Глава 8. Поток и диспетчеризация ввода/вывода.....</b>	<b>226</b>
Основные типы запросов ввода/вывода .....	227
Запросы на создание.....	227
Запросы на очистку и закрытие.....	227
Запросы на чтение и запись .....	228
Запросы управления вводом/выводом устройства .....	228
Обзор типов запросов ввода/вывода.....	230
Типы передач ввода/вывода .....	230
Буферизованный ввод/вывод.....	232
Прямой ввод/вывод .....	232
Ввод/вывод типа <i>METHOD_NEITHER</i> .....	232
Тип ввода/вывода <i>METHOD_NEITHER</i> в UMDF .....	232
Тип ввода/вывода <i>METHOD_NEITHER</i> в KMDF .....	233
Поток запросов ввода/вывода .....	234
Путь запроса ввода/вывода через стек устройств UMDF .....	235
Путь запросов ввода/вывода сквозь драйвер KMDF .....	237
Обработка завершения запросов ввода/вывода .....	239
Обработка завершения запросов ввода/вывода UMDF .....	239
Обработка завершения запросов ввода/вывода KMDF.....	239
Обработка завершения запросов ввода/вывода в Windows .....	240
Поток запросов ввода/вывода в инфраструктурах.....	240
Предварительная обработка пакетов IRP .....	242
Маршрутизация пакетов IRP к внутреннему обработчику запросов .....	242
Ход операций в обработчике запросов ввода/вывода .....	242
Проверка достоверности параметров .....	243
Обработка запросов в обработчике ввода/вывода.....	243
Объекты запросов ввода/вывода.....	244
Буферы ввода/вывода и объекты памяти.....	245
Извлечение буферов в драйверах UMDF.....	246
Извлечение буферов в драйверах KMDF.....	247
Время жизни запросов, памяти и указателей на буферы .....	253
Очереди ввода/вывода .....	253
Конфигурация очереди и типы запросов.....	254
Указание типа запроса для очереди.....	255
Обратные вызовы для очереди.....	255
Стандартные очереди.....	256
Очереди и управление энергопотреблением .....	257
Управляемые энергопотреблением очереди .....	257
Неуправляемые энергопотреблением очереди.....	259
Тип диспетчеризации .....	259
Управление очередью .....	260
Пример UMDF: создание очередей ввода/вывода .....	261
Стандартная очередь UMDF.....	262
Нестандартные очереди UMDF .....	263
Очереди UMDF с ручной диспетчеризацией .....	264
Пример KMDF: создание очередей ввода/вывода .....	265
Стандартная очередь KMDF.....	266
Нестандартная очередь KMDF .....	266
Извлечение запросов из очереди с ручной диспетчеризацией .....	267
Функции обратного вызова по событию.....	271
Объект файла для ввода/вывода.....	271
Автоматическая пересылка запросов на создание, очистку и закрытие .....	272
Обратные вызовы по событиям ввода/вывода для запросов на создание .....	273
Обработка запросов на создание в драйверах UMDF .....	274

Имперсонация в драйверах UMDF .....	275
Обработка запросов на создание в драйверах KMDF .....	278
Обратные вызовы по событиям ввода/вывода для очистки и закрытия .....	280
Функции обратного вызова для запросов на чтение, запись и IOCTL.....	281
Обратные вызовы для запросов на чтение и запись .....	282
Обратные вызовы для запросов IOCTL .....	286
Стандартные обратные вызовы для ввода/вывода .....	288
Завершение запросов ввода/вывода.....	290
Отмененные и приостановленные запросы.....	292
Отмена запросов ввода/вывода .....	292
Приостановка запроса ввода/вывода .....	294
Адаптивные тайм-ауты в UMDF .....	295
Самоуправляемый ввод/вывод .....	296
Самоуправляемый ввод/вывод при запуске и перезапуске устройства.....	298
Самоуправляемый ввод/вывод при переходе устройства в состояние пониженного энергопотребления или при его удалении.....	298
Пример KMDF: реализация таймера WDT.....	299
Пример: регистрация функций обратного вызова для самоуправляемого ввода/вывода .....	300
Пример: создание и инициализация таймера.....	300
Пример: запуск таймера.....	301
Пример: остановка таймера.....	302
Пример: перезапуск таймера .....	303
Пример: удаление таймера.....	303
<b>Глава 9. Получатели ввода/вывода .....</b>	<b>304</b>
О получателях ввода/вывода.....	305
Стандартные получатели ввода/вывода.....	305
Удаленные получатели ввода/вывода в драйверах KMDF .....	305
Общие и специализированные получатели ввода/вывода.....	306
Реализация получателя ввода/вывода в UMDF .....	307
Диспетчеры ввода/вывода UMDF .....	307
Внутристековые файлы для получателей ввода/вывода в драйверах UMDF .....	309
Создание и управление получателями ввода/вывода .....	310
Обращение к получателю ввода/вывода.....	310
Создание удаленных получателей ввода/вывода в драйверах KMDF .....	311
Функции инициализации для структуры параметров получателя ввода/вывода .....	311
Пример KMDF: создание и открытие удаленного получателя ввода/вывода .....	314
Управление состояниями получателя ввода/вывода .....	315
Методы для управления состоянием получателя ввода/вывода.....	316
Обратные вызовы получателей ввода/вывода для драйверов KMDF .....	317
Пример KMDF: функция обратного вызова <i>EvtIoTargetQueryRemove</i> .....	318
Создание запроса ввода/вывода.....	319
Пример UMDF: создание объекта запроса ввода/вывода WDF.....	319
Пример KMDF: создание объекта запроса ввода/вывода WDF.....	320
Объекты памяти и буферы для созданных драйвером запросов ввода/вывода .....	320
Выделение объектов памяти и буферов для запросов ввода/вывода .....	322
Родитель объекта памяти.....	322
Типы буферов .....	323
Одновременное создание объекта памяти и буфера.....	323
Создание объекта памяти, использующего уже существующий буфер.....	324
Сопоставление объекта памяти объекту запроса ввода/вывода .....	324
Пример UMDF: создание объекта памяти, использующего уже существующий буфер.....	325
Пример KMDF: создание объекта памяти и нового буфера .....	326

---

Форматирование запросов ввода/вывода.....	327
Когда нужно форматировать запрос.....	327
Форматирование запроса.....	327
Форматирование неизмененного запроса для стандартного получателя ввода/вывода.....	328
Форматирование измененных или созданных драйвером запросов .....	328
Методы форматирования UMDF для запросов ввода/вывода.....	328
Методы форматирования KMDF для запросов ввода/вывода.....	328
Параметры для методов форматирования.....	329
Пример UMDF: форматирование запроса на запись .....	329
Пример KMDF: форматирование запроса на чтение.....	330
Обратные вызовы для завершения ввода/вывода .....	331
Обработка в функции обратного вызова завершения ввода/вывода.....	331
Извлечение статуса завершения и другой информации.....	332
Отправление запросов ввода/вывода.....	333
Опция для отправки запросов .....	334
Значения тайм-аута для запросов ввода/вывода .....	334
Синхронные и асинхронные запросы ввода/вывода.....	335
Эффект, оказываемый состоянием получателя ввода/вывода .....	335
Опция "отправил и забыл" .....	336
Пример UMDF: отправка запроса стандартному получателю ввода/вывода .....	336
Пример KMDF: опция "отправил и забыл" .....	338
Пример KMDF: форматирование и отправление запроса получателю ввода/вывода .....	340
Разбивка запросов ввода/вывода на подзапросы .....	341
Пример KMDF: повторное использование объекта ввода/вывода .....	342
Отмена отправленного запроса .....	345
Пример UMDF: отмена всех запросов ввода/вывода для файла .....	346
Пример KMDF: отмена запросов ввода/вывода.....	347
Получатели ввода/вывода FileHandle в драйверах UMDF.....	347
Получатели ввода/вывода USB .....	349
Устройства USB .....	349
Конфигурационные дескрипторы и дескрипторы устройств .....	350
Модели передачи данных USB.....	352
Специализированные получатели ввода/вывода USB в WDF .....	353
Объекты устройств исполнителей USB .....	354
Объекты интерфейса USB .....	354
Объекты каналов исполнителей USB .....	355
Конфигурирование получателей ввода/вывода USB .....	355
Пример UMDF: конфигурирование получателя ввода/вывода USB.....	356
Пример KMDF: конфигурирование получателя ввода/вывода USB.....	359
Отправление запроса ввода/вывода получателю USB .....	363
Пример UMDF: отправление синхронного запроса получателю ввода/вывода USB.....	363
Пример KMDF: отправление асинхронного запроса ввода/вывода получателю USB .....	365
Средство непрерывного считывания USB в KMDF .....	368
Рекомендация для отправления запросов ввода/вывода.....	369
<b>Глава 10. Синхронизация.....</b>	<b>371</b>
Когда требуется применение синхронизации.....	372
Пример синхронизированного доступа к общим данным .....	372
Требования синхронизации для драйверов WDF .....	374
Возможности синхронизации, предоставляемые WDF .....	375
Подсчет ссылок и иерархическая объектная модель.....	376
Сериализация обратных вызовов функций для событий Plug and Play и энергопотребления.....	376
Управление потоком для очередей ввода/вывода.....	377
Блокировка представления объекта .....	377

Область синхронизации и сериализация функций обратных вызовов ввода/вывода.....	378
Область синхронизации на уровне устройства и методы диспетчеризации очереди.....	379
Область синхронизации в драйверах UMDF.....	382
Область синхронизации в драйверах KMDF.....	383
Стандартные настройки области синхронизации.....	385
Синхронизация на уровне устройства .....	385
Синхронизация на уровне очереди .....	385
Пример KMDF: область синхронизации .....	386
Автоматическая сериализация функций обратных вызовов процедурами DPC, таймерами и рабочими элементами .....	387
Уровни исполнения в драйверах KMDF.....	388
Спин-блокировки и wait-блокировки KMDF.....	390
Wait-блокировки.....	390
Спин-блокировки.....	391
Виды спин-блокировок .....	392
Пример KMDF: спин-блокировки.....	392
Синхронизация отмены запросов ввода/вывода в драйверах KMDF .....	394
Синхронизация отмены через область синхронизации.....	394
Синхронизация отмены с отслеживанием состояния в области контекста.....	395
Отмена входящих запросов синхронно с подзапросами.....	399
Сводка методов синхронизации и общие советы.....	401
<b>Глава 11. Трассировка и диагностируемость драйверов.....</b>	<b>403</b>
Основы программной трассировки с применением WPP .....	404
Преимущества программной трассировки WPP .....	404
Гибкое динамическое управление.....	404
Возможность просмотра сообщений в режиме реального времени или запись их в файл .....	404
Обильная информация .....	404
Охрана интеллектуальной собственности .....	405
Легкость перехода от операторов печати отладчика.....	405
Включение в поставляемые продукты .....	405
Минимальное воздействие на производительность .....	405
Компоненты программной трассировки WPP .....	405
Поставщик трассировки.....	406
Контроллер трассировки.....	406
Буфер трассировки .....	406
Сеанс трассировки.....	406
Потребитель трассировки .....	407
WPP и ETW.....	408
ETW в Windows Vista.....	408
Функции и макросы для работы с сообщениями трассировки.....	409
Макрос <i>DoTraceMessage</i> .....	410
Преобразование операторов печати отладчика в ETW .....	410
Условия для сообщений.....	410
Специальные функции сообщений трассировки .....	411
Поддержка программной трассировки в драйвере.....	411
Модифицирования файла Sources для исполнения препроцессора WPP .....	412
Пример UMDF: директива <i>RUN_WPP</i> для образца драйвера <i>Fx2_Driver</i> .....	413
Специальное сообщение трассировки драйвера <i>Fx2_Driver</i> .....	413
Пример KMDF: директива <i>RUN_WPP</i> для образца драйвера <i>Osrusbfx2</i> .....	414
Подключение TMH-файла .....	415
Пример UMDF: подключение TMH-файла .....	415
Пример KMDF: подключение TMH-файла .....	415

---

Определение контрольного GUID и флагов.....	416
Пример UMDF: определение контрольного GUID и флагов.....	417
Пример KMDF: определение <i>WPP_CONTROL_GUIDS</i> и флагов .....	418
Инициализация и очистка трассировки.....	418
Инициализация трассировки .....	418
Очистка после трассировки .....	420
Оснащение кода драйвера.....	422
Пример UMDF: добавление вызовов функций сообщений трассировки в код драйвера.....	422
Пример KMDF: добавление вызовов функций сообщений трассировки в код драйвера.....	422
Инструменты для программной трассировки.....	423
Проведение сеанса программной трассировки.....	424
Подготовка драйвера.....	424
Просмотр журнала трассировки драйвера утилитой TraceView.....	425
Создание и просмотр файла журнала трассировки .....	425
Просмотр журнала трассировки в режиме реального времени .....	426
Просмотр инфраструктурного журнала трассировки с помощью базовых инструментов трассировки.....	427
Практические рекомендации: думайте о диагностике .....	429
<b>Глава 12. Вспомогательные объекты WDF .....</b>	<b>431</b>
Выделение памяти.....	431
Локальное хранилище .....	432
Объекты памяти и буферы ввода/вывода .....	432
Объекты памяти UMDF и их интерфейсы.....	433
Объекты памяти KMDF и их методы.....	434
Обращение к реестру .....	436
Хранилище свойств устройства UMDF .....	436
Объекты реестра KMDF и их методы.....	439
Общие объекты .....	443
Пример UMDF: создание общего объекта .....	443
Пример KMDF: создание общего объекта .....	444
Объекты коллекции KMDF .....	444
Методы коллекции .....	445
Пример: создание и использование коллекции.....	446
Объекты таймера KMDF .....	448
Методы объекта таймера .....	448
Временные интервалы.....	449
Функция обратного вызова <i>EvtTimerFunc</i> .....	450
Пример: использование объекта таймера .....	450
Поддержка интерфейса WMI в драйвере KMDF .....	452
Работа с WMI .....	452
Требования для поддержки WMI .....	453
Инициализация поддержки WMI .....	454
Ресурс MOF.....	454
Объект поставщика WMI .....	454
Объекты экземпляра WMI .....	455
Функции обратного вызова для событий экземпляра WMI .....	458
Пример: запрос данных у экземпляра WMI .....	458
Пример: организация экземпляра WMI .....	459
Пример: организация элемента данных WMI .....	460
<b>Глава 13. Шаблон UMDF-драйвера .....</b>	<b>462</b>
Описание образца драйвера Skeleton.....	462
Образец драйвера Skeleton.....	463

Файлы образца драйвера Skeleton .....	463
Исходные файлы .....	463
Файлы поддержки компоновки .....	464
Файлы для поддержки инсталляции .....	464
Модификация файлов образца драйвера Skeleton под собственные требования .....	464
Инфраструктура DLL .....	465
Функция <i>DllMain</i> .....	465
Функция <i>DllGetClassObject</i> .....	466
Базовая поддержка технологии COM .....	467
Класс <i>CUnknown</i> .....	467
Класс <i>CClassFactory</i> .....	467
Объект обратного вызова для образца драйвера Skeleton .....	468
Метод <i>CreateInstance</i> .....	468
Интерфейс <i>IUnknown</i> .....	469
Интерфейс <i>IDriverEntry</i> .....	469
Необязательные интерфейсы .....	470
Объект обратного вызова устройства в образце драйвера Skeleton .....	470
Вспомогательные методы для объекта обратного вызова устройства .....	471
Модификация вспомогательных файлов образца драйвера Skeleton .....	473
Файл Sources .....	473
Файлы Make .....	474
Файл Exports .....	474
Файл версии ресурсов .....	475
Файл INX .....	475
<b>ЧАСТЬ IV. СМОТРИМ ГЛУБЖЕ — БОЛЬШЕ О ДРАЙВЕРАХ WDF .....</b>	<b>481</b>
<b>Глава 14. За пределами инфраструктур .....</b>	<b>483</b>
Использование системных сервисов, не входящих в состав инфраструктур .....	483
Использование функций Windows API в драйверах UMDF .....	484
Для драйверов UMDF, исполняющихся на Windows XP .....	486
Применение вспомогательных процедур режима ядра в драйверах KMDF .....	486
Обработка запросов, не поддерживаемых инфраструктурами .....	489
Обработка по умолчанию неподдерживаемых запросов .....	489
Обработка неподдерживаемых запросов в драйверах KMDF .....	490
Пример: организация функции обратного вызова предварительной обработки <i>EvtDriverDeviceAdd</i> .....	491
Пример: обработка в функции обратного вызова <i>EvtDeviceWdmPreprocessIrp</i> .....	491
<b>Глава 15. Планирование, контекст потока и уровни IRQL .....</b>	<b>493</b>
Потоки .....	494
Планирование потоков .....	494
Определение контекста потока .....	495
Контекст потока драйверных функций KMDF .....	496
Уровни IRQL .....	497
Уровни IRQL, специфичные для процессора и потока .....	498
Уровень IRQL PASSIVE_LEVEL .....	499
Уровень IRQL PASSIVE_LEVEL в критической области .....	500
Уровень IRQL APC_LEVEL .....	500
Уровень IRQL DISPATCH_LEVEL .....	500
Уровень IRQL DIRQL .....	502
Уровень IRQL HIGH_LEVEL .....	502
Рекомендации по исполнению на уровне IRQL DISPATCH_LEVEL или высшем .....	503
Вызовы функций, исполняющихся на низких уровнях IRQL .....	503

---

Сценарии прерывания потоков .....	504
Прерывание потока на однопроцессорной системе.....	504
Прерывание потока на многопроцессорной системе .....	505
Тестирование на наличие проблем, связанных с IRQL .....	507
Способы получения текущего уровня IRQL .....	507
Макросы <i>PAGED_CODE</i> и <i>PAGED_CODE_LOCKED</i> .....	508
Опции инструмента Driver Verifier .....	508
Рабочие элементы и драйверные потоки.....	509
Рабочие элементы.....	510
Пример KMDF: применение рабочего элемента .....	510
Оптимальные методики для управления контекстом потока и IRQL в драйверах KMDF .....	512
<b>Глава 16. Аппаратные ресурсы и прерывания.....</b>	<b>514</b>
Аппаратные ресурсы.....	514
Идентификация и освобождение аппаратных ресурсов .....	515
Идентификация ресурсов: подготовка аппаратного обеспечения.....	516
Освобождение аппаратного обеспечения.....	516
Списки ресурсов .....	517
Анализ аппаратных ресурсов драйвером.....	517
Платформенная независимость и отображение ресурсов драйвера .....	518
Пример: отображение ресурсов .....	519
Пример: отмена отображения ресурсов.....	522
Прерывания и их обработка .....	522
Объекты прерывания.....	523
Структура конфигурации объекта прерывания.....	524
Разрешение и запрещение прерываний .....	526
Обработка после разрешения и до запрещения прерываний.....	528
Процедуры обслуживания прерываний.....	529
Отложенная обработка прерываний .....	531
Синхронизация обработки на уровне DIRQL .....	532
<b>Глава 17. Прямой доступ к памяти.....</b>	<b>533</b>
Базовые понятия и терминология DMA.....	533
Транзакции DMA и передачи DMA .....	534
Пакетный DMA и DMA с применением общего буфера.....	534
Устройства DMA пакетной конструкции .....	535
Устройства DMA с применением общего буфера .....	535
Устройства DMA гибридной конструкции .....	535
Поддержка метода "разбиение/объединение" .....	535
Информация об устройстве, специфичная для DMA.....	536
Информация об устройстве и конструкция драйверов DMA .....	537
Тип конструкции DMA .....	537
Возможности адресации устройства.....	538
Аппаратная поддержка метода "разбиение/объединение" .....	538
Максимальный объем данных передачи.....	538
Требуемое выравнивание буфера.....	538
Факторы, не принимающиеся к рассмотрению .....	538
Абстракция DMA в Windows .....	539
Операции DMA и кэш процессора.....	540
Завершение передач DMA сбросом кэшей.....	541
Регистры отображения.....	541
Концепция.....	541
Реализация .....	543
Когда применять регистры отображения .....	543

---

Поддержка системного механизма "разбиение/объединение" .....	544
Концепция .....	544
Реализация .....	545
Передача DMA по любому адресу физической памяти .....	545
Концепция .....	546
Реализация .....	546
Реализация драйверов DMA.....	547
Инициализации драйвера DMA.....	548
Объект выключателя DMA.....	549
Объект общего буфера.....	549
Объект транзакций DMA .....	550
Пример: инициализации драйвера DMA .....	550
Инициирование транзакции.....	552
Инициализация транзакции .....	552
Исполнение транзакции .....	553
Пример: инициирование транзакции .....	553
Обработка запроса.....	555
Определение функции <i>EvtProgramDma</i> .....	555
Задачи, выполняемые функцией <i>EvtProgramDma</i> .....	556
Пример: обработка запроса .....	557
Обработка завершения DMA.....	558
Завершение передачи, транзакции и запроса.....	558
Пример: обработка завершения DMA .....	559
Тестирование драйверов DMA.....	560
Верификация, специфичная для DMA.....	560
Расширение отладчика <i>!dma</i> .....	561
Расширения отладчика KMDF для DMA .....	561
Рекомендации к разработке драйверов DMA .....	563
<b>Глава 18. Введение в COM .....</b>	<b>564</b>
Прежде чем приступить.....	564
Структура драйвера UMDF .....	565
Краткий обзор COM .....	567
Содержимое объекта COM .....	568
Объекты и интерфейсы .....	568
Интерфейс <i>IUnknown</i> .....	569
Подсчет ссылок.....	570
Рекомендации по использованию методов <i>AddRef</i> и <i>Release</i> .....	570
Подсчет ссылок .....	570
Использование указателей интерфейса в качестве параметров.....	570
Вызовы метода <i>Release</i> .....	571
Исправление ошибок со счетчиком ссылок .....	571
Идентификаторы GUID.....	572
Таблица VTables .....	572
<i>HRESULT</i> .....	573
Свойства и события .....	574
Библиотека ATL.....	575
Файлы IDL.....	575
Использование объектов COM UMDF .....	577
Использование объекта COM .....	577
Получение интерфейса через метод обратного вызова.....	578
Получение интерфейса через создание объекта UMDF .....	578
Получение интерфейса через вызов метода <i>QueryInterface</i> .....	579
Управление временем жизни объекта COM.....	580

---

Реализация инфраструктуры DLL .....	580
Функция <i>DllMain</i> .....	581
Функция <i>DllGetClassObject</i> .....	582
Фабрика классов .....	583
Реализация фабрики классов .....	583
Объекты, не требующие фабрики классов .....	585
Реализация объектов обратного вызова UMDF .....	586
Реализация класса для объекта COM .....	586
Реализация интерфейса <i>IUnknown</i> .....	587
Методы <i>AddRef</i> и <i>Release</i> .....	588
Функция <i>QueryInterface</i> .....	589
Реализация объектов обратного вызова UMDF .....	591
<b>ЧАСТЬ V. СОЗДАНИЕ, УСТАНОВКА И ТЕСТИРОВАНИЕ ДРАЙВЕРОВ WDF .....</b>	<b>593</b>
<b>Глава 19. Сборка драйверов WDF.....</b>	<b>595</b>
Общие положения по сборке драйверов .....	595
Драйверы UMDF: обстоятельства, учитывающиеся при сборке .....	596
Драйверы KMDF: обстоятельства, учитывающиеся при сборке .....	596
Введение в сборку драйверов .....	597
Среда сборки .....	597
Вспомогательные файлы утилиты Build .....	598
Обязательные файлы .....	599
Необязательные файлы .....	599
Ограничения на файлы проекта .....	600
Сборка проекта .....	600
Широко употребляемые флаги утилиты Build .....	601
Распространенные выходные файлы, создаваемые утилитой Build .....	601
Полезные советы для UMDF .....	602
Пример UMDF: сборка образца драйвера Fx2_Driver .....	602
Файл Sources для драйвера Fx2_Driver .....	602
Макросы файла Sources для драйвера Fx2_Driver .....	603
Номера версии UMDF .....	603
Стандартные выходные файлы .....	604
Специальные выходные файлы .....	604
Исходные файлы, заголовки и библиотеки .....	604
Конфигурация сборки .....	605
Файлы Makefile и Makefile.inc для драйвера Fx2_Driver .....	606
Сборка драйвера Fx2_Driver .....	606
Пример KMDF: сборка образца драйвера Osrusbf2 .....	607
Файл Sources для драйвера Osrusbf2 .....	607
Макросы файла Sources для драйвера Osrusbf2 .....	608
Номер версии KMDF .....	608
Стандартные выходные файлы .....	608
Специальные выходные файлы .....	608
Исходные файлы, заголовки и библиотеки .....	608
Конфигурация сборки .....	609
Файлы Makefile и Makefile.inc для драйвера Osrusbf2 .....	609
Сборка драйвера Osrusbf2 .....	610
<b>Глава 20. Установка драйверов WDF.....</b>	<b>611</b>
Основы установки драйверов .....	612
Ключевые задачи по установке драйверов .....	612
Инструменты и методы установки .....	612

Аспекты, принимаемые во внимание при установке драйверов WDF .....	613
Управление версиями WDF и установка драйверов.....	613
Обновления дополнительных версий .....	614
Обновления основных версий .....	614
Распространение инфраструктуры.....	614
Привязка драйверов к инфраструктуре .....	615
Драйверы KMDF .....	615
Драйверы UMDF .....	616
Пакеты соинсталляторов WDF .....	616
Пакет соинсталлятора UMDF .....	617
Пакет соинсталлятора KMDF .....	618
Компоненты драйверного пакета WDF .....	618
Создания INF-файла для драйверного пакета WDF .....	619
Широко применяемые разделы INF-файла .....	620
Инструменты для работы с INF-файлами.....	621
Файлы INF для разных процессорных архитектур .....	621
Файлы INF драйверов WDF: разделы соинсталляторов .....	622
Примеры INF-файлов WDF .....	624
Пример UMDF: INF-файл драйвера Fx2_Driver .....	625
Пример KMDF: INF-файл драйвера Osrusbfx2 .....	628
Подписание и распространение драйверного пакета.....	629
Подписанные файлы каталогов .....	629
Указание файла каталогов в INF-файле .....	630
Подписывание драйверов BSD.....	630
Распространение драйверного пакета .....	630
Установка драйверов .....	631
Факторы, принимаемые во внимание для тестовых установок .....	631
Факторы, принимаемые во внимание для эксплуатационных установок.....	632
Установка драйверов с помощью менеджера PnP .....	632
Установка драйверов с помощью SPInst или DIFxApp.....	633
Установка драйверов с помощью специализированного установочного приложения .....	634
Установка и обновление драйверов с помощью инструмента DevCon .....	634
Обновление драйверов с помощью Диспетчера устройств .....	635
Удаление драйверов .....	635
Процесс установки драйверов .....	636
Этап 1. Установка требуемой инфраструктуры WDF (если необходимо).....	636
Этап 2. Создание узла devnode для устройства.....	636
Этап 3. Развёртывание драйверного пакета в хранилище драйверов .....	636
Этап 4. Установка драйвера.....	636
Действия по удалению драйвера .....	636
Удаление устройства .....	637
Удаление драйверного пакета из хранилища драйверов.....	637
Удаление двоичных файлов драйвера .....	637
Инструменты для удаления устройств и драйверов .....	638
Поиск и удаление проблем с установкой драйверов WDF .....	639
Поиск и исправление ошибок установки с помощью отладчика WinDbg.....	639
Журналы регистрации ошибок установки драйверов .....	640
Распространенные ошибки при установке драйверов WDF .....	641
Фатальные ошибки при установке .....	641
Коды ошибок менеджера PnP .....	642
<b>Глава 21. Инструменты для тестирования драйверов WDF .....</b>	<b>644</b>
Начало работы по тестированию драйверов.....	645
Выбор тестовой системы .....	645

---

Обзор инструментов для тестирования драйверов WDF .....	646
Методы трассировки для тестирования драйверов .....	646
Инструменты PREfast и SDV .....	647
Другие инструменты для тестирования драйверов.....	647
Инструмент INF File Syntax Checker (ChkINF) .....	648
Инструмент Device Console .....	648
Инструмент Device Path Exerciser .....	649
Инструменты Kern Rate и Kern Rate Viewer.....	649
Инструмент Plug and Play Driver Test.....	650
Инструмент Plug and Play CPU Test.....	651
Инструмент Memory Pool Monitor .....	651
Инструмент Power Management Test Tool .....	651
Инфраструктура WDTF.....	652
Инструмент Driver Verifier .....	654
Когда использовать Driver Verifier .....	654
Как работает Driver Verifier.....	655
Как работать с Driver Verifier.....	656
Использование утилиты Verifier из командной строки.....	656
Запуск Driver Verifier Manager .....	657
Примеры работы с Driver Verifier .....	657
Пример 1: активирование стандартных опций для нескольких драйверов .....	657
Пример 2: активирование специальных опций для всех драйверов.....	657
Пример 3: запуск и остановка проверки драйвера без перезагрузки .....	658
Пример 4: активирование или деактивирование опций без перезагрузки.....	659
Пример 5: деактивирование всех опций Driver Verifier .....	659
Пример 6: деактивирование Driver Verifier .....	659
Пример 7: эмуляция недостаточных ресурсов .....	659
Пример 8: принудительная проверка уровня IRQL (Force IRQL Checking) .....	661
Использование информации от Driver Verifier при отладке .....	662
Пример 1: просмотр трассировок операций стека с помощью <i>!Verifier</i> .....	662
Пример 2: использование <i>!Verifier</i> для вывода счетчиков ошибок и выделений памяти из пула.....	663
Инструмент KMDF Verifier .....	664
Когда использовать KMDF Verifier .....	665
Как работает KMDF Verifier.....	665
Как активировать KMDF Verifier .....	665
Использование информации от KMDF Verifier при отладке .....	666
Верификатор UMDF Verifier .....	667
Остановы <i>bugcheck</i> UMDF .....	668
Извещение об ошибках UMDF.....	669
Верификатор Application Verifier.....	669
Как работает Application Verifier .....	670
Использование Application Verifier для верификации драйверов UMDF .....	670
Оптимальные методики для тестирования драйверов WDF.....	671
Советы по сборке драйверов .....	671
Советы по использованию инструментов наилучшим образом .....	672
Советы для тестирования драйвера на протяжении его жизненного цикла.....	673
<b>Глава 22. Отладка драйверов WDF .....</b>	<b>674</b>
Обзор инструментов отладки для WDF .....	674
Отладчик WinDbg.....	675
Прочие инструменты.....	675
Трассировка WPP .....	676
Отладка макросов и процедур .....	676

Основы отладчика WinDbg .....	676
Проверочные и свободные версии сборок .....	677
Пользовательский интерфейс .....	677
Команды отладчика .....	678
Символы и исходный код .....	679
Расширения отладчика .....	681
Подготовка к отладке драйверов UMDF .....	682
Как разрешить отладку кода загрузки и запуска драйвера .....	683
Как начать отладку кода загрузки и запуска драйвера UMDF .....	684
Отладка исполняющегося драйвера UMDF .....	685
Отслеживание объектов UMDF и подсчет ссылок .....	686
Отладка фатального сбоя драйвера UMDF .....	687
Подготовка к отладке драйверов KMDF .....	687
Как активировать отладку режима ядра на тестовом компьютере .....	688
Как активировать отладку режима ядра для Windows Vista .....	688
Как активировать отладку режима ядра для более ранних, чем Windows Vista, версий Windows .....	689
Подготовка тестового компьютера к отладке драйверов KMDF .....	689
Как начать сеанс отладки KMDF .....	690
Как начать отладку фатального сбоя драйвера KMDF .....	692
Пошаговый разбор отладки драйверов UMDF на примере образца драйвера Fx2_Driver .....	692
Подготовка к сеансу отладки драйвера Fx2_Driver .....	692
Начало сеанса отладки для драйвера Fx2_Driver .....	693
Анализ процедуры обратного вызова <i>OnDeviceAdd</i> для драйвера Fx2_Driver .....	693
Исследование с помощью расширений отладчика UMDF объекта обратного вызова устройства .....	695
Исследование с помощью расширений отладчика UMDF запроса ввода/вывода .....	696
Пошаговый разбор отладки драйверов KMDF на примере образца драйвера Osrusbfx2 .....	697
Подготовка к сеансу отладки драйвера Osrusbfx2 .....	698
Начало сеанса отладки для драйвера Osrusbfx2 .....	698
Анализ процедуры обратного вызова <i>EvtDriverDeviceAdd</i> .....	698
Исследование объекта устройства с помощью расширений отладчика KMDF .....	699
Исследование запроса ввода/вывода с помощью расширений отладчика UMDF запроса ввода/вывода .....	700
Просмотр сообщения трассировки с помощью WinDbg .....	702
Просмотр журнала KMDF с помощью WinDbg .....	703
Получение информации протоколирования после останова bugcheck .....	704
Управление содержимым журнала KMDF .....	705
Дополнительные предложения для экспериментирования с WinDbg .....	706
<b>Глава 23. Инструмент PREfast for Drivers .....</b>	<b>707</b>
Введение в PREfast .....	707
PREfast и инструмент Code Analysis для Visual Studio .....	708
Как работает PREfast .....	708
Какие ошибки может выявлять PREfast .....	709
Использование PREfast .....	710
Задание режима анализа для PREfast .....	710
Как запустить PREfast .....	710
Сборка примеров PREfast .....	711
Вывод на экран результатов анализа PREfast .....	713
Утилита просмотра журнала дефектов, обнаруженных PREfast .....	713
Полезные советы для фильтрации результатов PREfast .....	716
Вывод журнала дефектов PREfast в текстовом виде .....	717

---

Примеры результатов анализа PREfast .....	718
Пример 1: неинициализированные переменные и нулевые указатели.....	718
Пример 2: неявный порядок вычислений.....	719
Пример 3: вызов функции на неправильном уровне IRQL .....	720
Пример 4: действительная ошибка, но указанная в неправильном месте .....	721
Пример 5: несоответствие класса типа функции .....	723
Пример 6: неправильный перечислимый тип .....	724
Практики кодирования, улучшающие результаты анализа PREfast .....	725
Предупреждения, указывающие распространенные причины шума и как на них реагировать .....	726
Воздействие вставленного кода ассемблера на результаты анализов PREfast .....	727
Использование директивы <i>pragma warning</i> для подавления шума.....	728
Использование аннотаций для устраниния шума .....	728
Использование аннотаций .....	729
Как аннотации улучшают результаты PREfast.....	729
Аннотации расширяют прототипы функций .....	730
Аннотации описывают контракт между вызываемой функцией и вызывающим ее клиентом.....	730
Аннотации помогают усовершенствовать конструкцию функции .....	730
Куда вставлять аннотации в коде .....	731
Аннотирование функций и параметров функций .....	732
Аннотации для объявлений <i>typedef</i> .....	733
Аннотации к объявлению <i>typedef</i> функций.....	735
Советы по размещению аннотаций в исходном коде .....	737
Аннотации общего назначения .....	737
Аннотирование входных и выходных параметров .....	738
Контракт аннотаций <i>_in</i> и <i>_out</i> .....	739
Аннотации <i>_in</i> , <i>_out</i> и <i>_inout</i> и макросы <i>IN</i> , <i>OUT</i> и <i>INOUT</i> .....	740
Модификаторы аннотаций.....	741
Модификатор <i>_opt</i> .....	741
Модификатор <i>_deref</i> .....	741
Аннотации размера буфера.....	742
Аннотации буфера неизменяемого размера .....	743
Сводка аннотаций для буферов .....	744
Полезные советы по аннотированию буферов.....	746
Примеры аннотаций буфера .....	746
Аннотации строк .....	747
Аннотация <i>_nullterminated</i> .....	747
Аннотация <i>_nullnullterminated</i> .....	748
Аннотация <i>_possibly_notnullterminated</i> .....	748
Зарезервированные параметры.....	749
Возвращаемые функциями значения .....	749
Аннотации для драйверов .....	750
Базовые аннотации и соглашения по их применению .....	753
Списки аннотаций для драйверов .....	753
Вложенные аннотации .....	754
Условные аннотации .....	755
Примеры вложенных условных аннотаций .....	756
Грамматика условных выражений .....	757
Аннотации результатов функций .....	758
Аннотации для сопоставления типов.....	760
Аннотации указателей.....	761
Аннотации для постоянных и переменных параметров .....	762
Аннотации форматирующих строк.....	762

---

Диагностические аннотации.....	763
Аннотации для предпочтительных функций.....	763
Аннотации для сообщений об ошибках .....	763
Аннотации для функций в операторах <i>_try</i> .....	764
Аннотации для памяти .....	764
Аннотации для выделения и освобождения памяти .....	764
Аннотации для совмещенных имен памяти .....	765
Аннотации для ресурсов иных, чем память .....	766
Аннотации для получения и освобождения ресурсов, иных, чем память .....	767
Аннотации для глобальных ресурсов, иных, чем память.....	767
Аннотации для критической области и спин-блокировки отмены .....	768
Составные аннотации для ресурсов.....	771
Аннотации класса типа функций.....	772
Аннотации для идентификации класса типа функции .....	773
Аннотации для проверки класса типа функции в условном выражении .....	773
Аннотации для плавающей запятой.....	774
Аннотации для уровней IRQL .....	775
Аннотации для указания максимального и минимального уровня IRQL .....	776
Аннотации для явного указания уровня IRQL.....	777
Аннотации для повышения и понижения уровня IRQL.....	777
Аннотации для сохранения и восстановления уровня IRQL .....	777
Аннотации для удерживания постоянного уровня IRQL.....	778
Аннотации для сохранения и восстановления уровня IRQL для процедур отмены ввода/вывода .....	778
Примеры аннотаций IRQL.....	779
Полезные советы по применению аннотаций IRQL.....	780
Аннотация <i>DO_DEVICE_INITIALIZING</i> .....	781
Аннотации для operandов с взаимоблокировкой .....	781
Примеры аннотированных системных функций.....	781
Составление и отладка аннотаций .....	784
Образцы контрольных примеров для аннотаций.....	784
Полезные советы для написания контрольных примеров для аннотаций .....	785
Оптимальные методики работы с PRE/fast .....	786
Оптимальные методики для PRE/fast.....	786
Оптимальные методики для использования аннотаций.....	787
Пример: аннотированный заголовочный файл Osrusbf2.h .....	789
<b>Глава 24. Инструмент Static Driver Verifier .....</b>	<b>794</b>
Введение в SDV .....	794
Принцип работы SDV .....	795
Правила SDV .....	796
Как SDV применяет правила к коду драйвера .....	798
А что внутри? Принцип работы движка верификации SDV.....	799
Абстракция.....	800
Поиск и проверка достоверности.....	800
Усовершенствование.....	801
Как аннотировать исходный код драйверов KMDF для SDV .....	801
Объявления типов ролей функций для драйверов KMDF .....	802
Пример: использование типов ролей функций в образцах драйверов .....	802
Использование SDV .....	804
Как подготовить файлы и выбрать правила для SDV .....	804
Подготовка к использованию SDV .....	804
Очистка каталога sources драйвера .....	805
Обработка библиотек, используемых драйвером .....	805
Как отсканировать исходный код для создания файла Sdv-map.h .....	806

Выполнение верификации .....	808
Экспериментирование с SDV .....	810
Просмотр отчетов SDV .....	811
Утилита Defect Viewer .....	813
Оптимальная методика: проверяйте результаты SDV.....	814
Правила KMDF для SDV .....	815
Правила последовательности функций DDI для KMDF .....	816
Правила инициализации устройств для KMDF .....	816
Правила очистки устройства управления для KMDF.....	817
Правила завершения запроса для KMDF.....	817
Правила отмены запроса для KMDF.....	818
Правила для буферов, списков MDL и памяти запросов .....	819
Правила для буферов запросов.....	820
Правила для запросов обращения к спискам MDL.....	820
Правила для запросов обращения к памяти .....	821
Правила DDI для владельцев политики энергопотребления .....	821
Пример: пошаговый анализ драйвера Fail_Driver3 с помощью SDV.....	823
Подготовка к верификации драйвера Fail_Driver3 .....	824
Сборка библиотеки драйвера Fail_Driver3 с помощью SDV.....	824
Создание файла Sdv-map.h для драйвера Fail_Driver3 .....	824
Выполнение верификации драйвера Fail_Driver3.....	825
Просмотр результатов верификации драйвера Fail_Driver3 .....	826
Ознакомление с нарушенными правилами.....	827
Пошаговый просмотр трассировки нарушения правила в драйвере Fail_Driver3 .....	827
Типы ролей функций обратного вызова KMDF для SDV .....	828
<b>Словарь .....</b>	<b>831</b>
<b>Предметный указатель.....</b>	<b>852</b>

# **Об авторах**

## **Пенни Орвик**

Пенни Орвик (Penny Orwick) — независимая писательница, работающая на компанию Steyer Associates и специализирующаяся в разработке драйверов и тематике операционной системы Windows. Ее первые статьи на тему драйверов Windows появились в 1997 г., и с самого начала проекта она работала с командой разработчиков Windows Driver Foundation, публикуя технические статьи о WDF для сообщества разработчиков драйверов. Пенни получила диплом бакалавра искусств (Bachelor of Arts) в Корнельском университете (Cornell University) и диплом магистра изобразительных искусств (Master of Fine Arts) в Университете штата Монтана (University of Montana).

## **Гай Смит**

Гай Смит (Guy Smith) начал программировать на языке Фортран IV с перфокартами вместо клавиатуры, в свою бытность аспирантом по геофизике. Он начал писать документацию SDK для корпорации Microsoft в 1996 г., и с тех пор работал над многими технологиями от Microsoft, включая Window CE, Windows Shell and Common Controls, DirectX 8, Internet Explorer и Windows Presentation Foundation. Сейчас он работает независимым писателем на компанию Steyer Associates, фокусируясь на драйверах устройств и связанной тематике программного обеспечения режима ядра. Гай — музыкант-любитель, увлекающийся музыкой средних веков и эпохи Возрождения, играющий на лютне, цитре и серпенте.

## **Кэрол Бухмиллер**

Кэрол Бухмиллер (Carol Buchmiller) пишет и редактирует документацию для программного обеспечения для персональных компьютеров, начиная с 1981 г. Кэрол работала с Microsoft в начале 80-х годов прошлого столетия, после чего она сотрудничала с различными компаниями-производителями программного обеспечения в штатах Орегон и Вашингтон, в конечном счете начав специализироваться в тематике драйверов режима ядра Windows и совместимости аппаратного обеспечения. В 2000 г. Кэрол начала работать в Microsoft писателем для Web-сайта Windows Hardware Developer Central. Кэрол получила диплом бакалавра

искусств в Колледже Уитмена (Whitman College) и завершила программы сертификации по языкам С и С++.

## **Энни Пирсон**

Энни Пирсон (Annie Pearson) начала писать и редактировать документацию для программного обеспечения в 1982 г. Первым опытом технической документации и управления проектами по написанию технической документации для Windows для нее была работа над документацией для Windows 3.1 Resource Kit. Энни работает в качестве ведущего писателя и информационного разработчика для Web-сайта Windows Hardware Developer Central с 1997 г.

# Предисловие

Одной из почестей, но также и хлопот, сопутствующих моей ответственности за разработку и поддержку основного компонента ядра операционной системы Microsoft Windows, является то, что у меня есть возможность анализировать множество сбоев операционной системы, возникающих в этом компоненте. Будучи ответственным за менеджер ввода/вывода, я имел возможность выяснить причины многих связанных с драйверами проблем. Анализируя эти сбои, я многому научился. В процессе анализа я начал различать определенные закономерности.

Я пришел к выводу, что для комплексного понимания этих проблем мне не хватало знаний о различных стеках устройств (таких как для работы с устройствами хранения, аудио и видео) и о межкомпонентных соединениях (таких как USB и 1394). Поэтому я запустил среди лидеров разработок (development leads) команд разработчиков устройств в подразделении Windows проект, называемый "Driver Stack Reviews" ("Анализы драйверных стеков"). После многочисленных анализов мы пришли к выводу, что наша базовая драйверная модель была слишком сложной. Наши абстракции были неправильными, и мы перекладывали слишком большую часть работы на разработчиков драйверов.

После 14 лет разработки драйверная модель Windows (Windows Driver Model, WDM) существенно разрослась, и ее возраст начинал давать о себе знать. Хотя модель WDM очень гибкая и может поддерживать множество различных устройств, уровень ее абстракции довольно низкий. Она была создана для небольшого числа разработчиков, которые либо обладали глубоким пониманием ядра Windows, либо могли консультироваться с разработчиками ядра. На то большое число разработчиков драйверов, которые в настоящее время исчисляются в тысячах, эта модель не рассчитывалась.

Слишком много правил было не очень понятно многим из этих разработчиков, т. к. их было очень трудно доходчиво объяснить. Фундаментальные изменения операционной системы, такие как поддержка Plug and Play и управление энергопотреблением, не были удовлетворительно интегрированы с подсистемой ввода/вывода Windows в большой степени потому, что мы хотели иметь возможность совместной работы драйверов, поддерживающих и не поддерживающих Plug and Play. Это означало, что конструкция операционной системы накладывала на драйверы громадную нагрузку по синхронизации событий Plug and Play и энергопотребления с запросами ввода/вывода. Правила для синхронизации были сложными, труднопонимаемыми и плохо задокументированными. Кроме этого, большинство драйверов не обрабатывало асинхронный ввод/вывод и отмену запросов ввода/вывода должным образом, хотя возможности асинхронного отменяемого ввода/вывода и были заложены в операционную систему с самого начала ее разработки.

Хотя эти заключения казались интуитивно очевидными, нам требовалось доказать их с применением внешних данных. Операционная система Windows XP содержит замечательную возможность, называющуюся Windows Error Reporting (WER, служба регистрации и сообщения об ошибках Windows), которая позволяет выполнять онлайновый анализ сбоев (Online Crash Analysis, OCA). При фатальном сбое системы создается дамп небольшого размера (minidump), который пользователи могут отправить в Microsoft для анализа. Когда мы увидели огромное число сбоев, то поняли, что нам нужно было фундаментальным образом изменить способ разработки драйверов.

Мы также провели исследование и лично беседовали со сторонними разработчиками драйверов, чтобы подтвердить достоверность наших заключений и представить наш план по упрощению драйверной модели. Эти встречи открыли нам глаза на многое. Большинство разработчиков драйверов считали нашу драйверную модель — а особенно компоненты, связанные с Plug and Play, управлением энергопотреблением и отменой асинхронных запросов — сложной и трудной в использовании. Разработчики выражали сильное желание иметь более простую драйверную модель. Кроме этого, они довели до нашего сведения несколько требований, которые мы сами раньше упустили из рассмотрения.

Прежде всего, более простая драйверная модель должна была поддерживать несколько операционных систем. Поставщики аппаратного обеспечения хотели иметь возможность создавать и поддерживать лишь один драйвер для нескольких версий операционной системы. Новая драйверная модель, которая могла бы работать только с самой последней версией Windows, была неприемлема.

Во-вторых, разработчики драйверов не желали быть ограниченными доступом только к небольшому набору функций API, что было нашим подходом с некоторыми специфичными для класса устройств драйверными моделями. Им требовалось иметь возможность выйти из пределов драйверной модели и работать на более низком уровне.

С этими доведенными к нашему сведению пожеланиями мы и приступили к работе над драйверной моделью Windows Driver Foundation (WDF). Нашей целью было создание драйверной модели следующего поколения, которая бы отвечала требованиям всех классов устройств.

Для WDF мы применили другую технологию разработки. Мы с самого начала вовлекли в разработку драйверной модели сторонних разработчиков драйверов, для чего мы проводили периодические оценки проекта. В ноябре 2000 г., как только мы разработали спецификации, мы пригласили нескольких разработчиков на их обсуждение за круглым столом. Таким образом, мы получили полезные комментарии еще перед тем, как начали написание кода. Мы открыли адрес электронной почты и дискуссионные группы, где обсуждали разные опции проекта. Несколько внутренних и внешних разработчиков приняло для своей работы раннюю версию нашей модели и предоставило нам исключительно полезные отзывы и замечания. Мы также запросили и получили отзывы и замечания от посетителей конференции WinHEC и разных групп новостей разработчиков драйверов.

Драйверная модель WDF подверглась нескольким переработкам, результатом которых является ее сегодняшняя версия. Принимая во внимание отзывы и замечания, полученные нами во время разработки, мы переделали реализацию Plug and Play и управления энергопотреблением, а также логику синхронизации. В частности, реализация Plug and Play и управления энергопотреблением была переделана, чтобы использовать машины состояний. Это помогло сделать операции явными, поэтому стало легко понимать взаимодействия между вводом/выводом и механизмом Plug and Play. По мере того, как разрабатывались новые драйверы WDF, мы открывали новые правила, связанные с Plug and Play и управлением

энергопотреблением, и включали эти правила в машины состояний. Одним из ключевых преимуществ от использования WDF есть то, что каждый драйвер автоматически получает копию этой хорошо продуманной и тщательно протестированной реализации Plug and Play и управления энергопотреблением.

Данные системы ОСА также указывали, что проблему со сбоями требовалось решать иным, более радикальным способом. Эти данные показывали, что 85% неожиданных остановов системы вызывалось драйверами, а не базовыми компонентами ядра Windows. Анализы показали, что драйверам для многих классов устройств — в особенности для соединений USB, Bluetooth и 1394 — не было необходимости исполняться в режиме ядра. Перемещение исполнения драйверов в пользовательский режим дает многие преимущества. Например, сбои драйверов пользовательского режима можно полностью изолировать, и нормальную работу системы можно восстановить без перезагрузки. Среда программирования в пользовательском режиме намного проще, чем в режиме ядра. Для написания кода разработчики имеют доступ ко многим инструментам и мощным языкам программирования. Выполнение отладки намного проще. WDF представляет значительное улучшение тем, что одна и та же драйверная модель предоставляется как в режиме ядра, так и в пользовательском режиме.

Хотя упрощения драйверной модели решают многие причины системных сбоев, они не решают проблемы ошибок программирования, таких как переполнение буфера, неинициализированные переменные, неправильное использование системных процедур и т. п. Работа в научно-исследовательском подразделении корпорации Microsoft, называющемся Microsoft Research (MSR), в области инструментов для статического анализа решила эту проблему. В подразделении MSR были разработаны прототипы инструментов, которые могут понимать правила драйверной модели и выполнять формальный анализ исходного кода. Мы решили воплотить две из этих идей в реальные инструменты, которые впоследствии стали частью WDF: инструменты Static Driver Verifier (SDV) и PREFast for Drivers (PFD).

С выпуском операционной системы Windows Vista, как первая версия WDF, так и наши инструменты статического анализа стали доступны разработчикам драйверов в наборе разработчика драйверов Windows Driver Kit (WDK). WDF и инструменты статического анализа заложили прочное основание для нашей платформы для разработки драйверов. Первоначальный выпуск Windows Vista содержал около 17 драйверов KMDF, охватывая широкий круг классов устройств. В пользовательском режиме драйверы UMDF поддерживаются как технологией Microsoft Windows Sideshow, так и технологией Windows Portable Media. Корпорация Microsoft будет продолжать развивать эту базу, чтобы обеспечить потребности текущих и будущих классов устройств.

В этой книге представляются основы инфраструктур модели WDF и инструментов статического анализа, впервые предоставляя один источник для всей связанной с WDF информации. Эта книга должна помочь разработчику драйверов любого уровня — даже начинающему — быстро войти в курс работы с WDF. Вы увидите, что WDF позволяет разрабатывать драйверы более высокого качества в значительно более короткий период времени, чем старые драйверные модели.

*Нар Ганапати (Nar Ganapathy)*  
Архитектор

*Группа Windows Device Experience Group*  
Корпорация Microsoft

# Благодарности

Авторы выражают глубокую благодарность за исключительный вклад в написание этой книги, сделанный членами команды Window Driver Foundation корпорации Microsoft путем предоставления технической информации, образцов кода, анализа и общей моральной поддержки. Дорон Холан (Doron Holan), Нарайнан Ганапати (Narayanan Ganapathy), Правин Рао (Praveen Rao), Ильяс Якуб (Eliyas Yakub) и Питер Вилант (Peter Wieland) играли важную роль в создании и формировании этой книги. Ключевой вклад также сделал Джон Ричардсон (John Richardson), создавая первые черновые версии. Дан Тэри (Donn Terry) и Влад Левин (Vlad Levin) предоставили важное руководство в разработке глав 23 и 24 соответственно.

Мы отдаляем должное лидерству архитектора проекта, Нарайана Ганапати, и его вкладу в ранние стадии разработки модели Windows Driver Foundation. Поддержку проекту в его ранних стадиях также предоставили такие одаренные богатым творческим воображением личности, как Брэд Карпентер (Brad Carpenter), Винс Оргован (Vince Orgovan), Боб Ринн (Bob Rinne), Роб Шорт (Rob Short) и Майк Трикер (Mike Tricker).

Руководитель проекта на стадии разработки WDF Иоган Мэриэн (Johan Marien) запустил в действие постоянно действующую программу по обсуждению направлений разработки драйверной модели с сообществом разработчиков драйверов для Windows. Мы выражаем признательность за значительный вклад, сделанный Питером Вискаролой (Peter Viscarola) и компанией Open Systems Resources, Inc. (OSR), особенно за их страстную увлеченность и за их работу над обучающим прибором OSR USB Fx2, на котором основано несколько образцов драйверов WDF. Ценные высказывания и замечательные идеи также были предоставлены многими наиболее ценными профессионалами (Most Valued Professionals) сообщества разработчиков драйверов и ведущими разработчиками драйверов для Windows: Даном Бурном (Don Burn), Тревором Говиасом (Trevor Goveas, компания Agere), Биллом Мак-Кензи (Bill McKenzie), Тимом Робертсом (Tim Roberts, компания Providenza & Boekelheide), Марком Радди (Mark Roddy, компания Hollis Technology Solutions), Эриком Тисот-Дюпоном (Eric Tissot-Dupont, компания Logitech) и Рэем Трентом (Ray Trent, компания Synaptics).

Такой обширный проект, как WDF, было бы невозможно осуществить без преданных своему делу команд разработчиков, усилиями которых идея была воплощена в реальный продукт.

Существование WDF было бы невозможным без исключительно талантливой группы проектирования и разработки. Значительный вклад в этом отношении сделали Робин Календар (Robin Callendar), Джо Дай (Joe Dai), Дорон Холан (Doron Holan), Вишал Манан (Vishal

Manan), Адриан Они (Adrian Oney), Джейк Ошинс (Jake Oshins), Рей Патрик (Ray Patrick), Гурупракаш РАО (Guruprakash Rao), Абхишек Рам (Abhishek Ram), Правин РАО (Praveen Rao), Джон Ричардсон (John Richardson), Мукунд Санкаранарайян (Mukund Sankaranarayyan), Эрик Смит (Erick Smith), Питер Вильт (Peter Wieland) и Ильяс Якуб (Eliyas Yakub).

Инструменты для статического анализа и верификации являются важной частью WDF. Том Бол (Tom Ball) и Сирам Раджамани (Sriram Rajamani) изобрели верификационную машину SLAM и с поддержкой группы разработчиков WDF сделали успешную презентацию Биллу Гейтсу по передовой технологии для верификации драйверов, основанной на SLAM. Эти события привели к запуску проекта по разработке инструмента Static Driver Verifier в группе разработчиков WDF. Впоследствии важный вклад в разработку инструментов статического анализа сделали Влад Левин (Vlad Levin), Дан Тери (Donn Terry), Элла Баумитова (Ella Baumitova), Байрон Кук (Byron Cook), Джекоб Лихтенберг (Jakob Lichtenberg) и Кон МакГарви (Con McGarvey).

Команда тестировщиков WDF обеспечивала качество разрабатываемых инфраструктур. Разработкой тестирования для WDF и инструментов статического анализа руководил Кетцел Бредли (Quetzel Bradley), для KMDF — Рави Голлапуди (Ravi Gollapudi), а для UMDF и инструментов статического анализа — Абдула Устунера (Abdullah Ustuner). Сотрудничали в тестировании KMDF Аруна Банда (Aruna Banda), Боб Кильгард (Bob Kjelgaard), Кумар Раджив (Kumar Rajeev) и Виллем ван дер Хувен (Willem van der Hoeven). В тестировании UMDF сотрудничали Шефали Гулати (Shefali Gulati), Шаямал Варма (Shyamal Varma), Джимми Чен (Jimmy Chen), Патрик Манингер (Patrick Maninger) и Джеймс Мой (James Moe). Джан Хаген (Jon Hagen), Онур Озыер (Onur Ozyer) и Джон Генри (John Henry) сотрудничали в тестировании инструментов статического анализа.

Критической составляющей для всех продуктов, а в особенности для инструментов разработчика, является документация. Важный вклад в разработку документации WDF сделали Ричард Браун (Richard Brown), Дэйв Хаген (Dave Hagen), Джон Джексон (John Jackson) и Адам Вильсон (Adam Wilson).

Мы выражаем свою признательность руководству корпорации Microsoft за веру и инвестиции в проект WDF, а также руководителям программ за помощь в идентификации и удалении препятствий на пути к доставке Windows Driver Foundation сообществу разработчиков драйверов. Проектом руководили Фрэн Дагерти (Fran Dougherty), Стю Фарнам (Stu Farnham) и Хариш Найду (Harish Naidu). Программами проекта руководили Иоган Мэриэн (Johan Marien), Джефри Коупленд (Jeffrey Copeland), Муртуза Нагутанавала (Murtuza Naguthanawala), Богуш Ондрушек (Bohus Ondrussek) и Тереза Стоун (Teresa Stone).

Многие команды корпорации Microsoft внесли свой вклад в программу WDF, приняв на свое вооружение ранние версии WDF и предоставляя команде разработчиков свои отзывы и рекомендации. А именно:

- ◆ команда Microsoft Hardware — Вадим Дмитриев (Vadim Dmitriev);
- ◆ команда Tablet PC — Микки Дуроджайе (Mikki Durojaiye);
- ◆ команда Windows Client Technologies для рынков развивающихся стран — Жангвей Жу (Zhangwei Xu);
- ◆ команда менеджера ввода/вывода Windows — Джон Ли (John Lee), Пол Сливович (Paul Sliwowicz);
- ◆ команда Windows Media Player/Windows Portable Devices — Орен Розенблум (Oren Rosenbloom), Влад Садовски (Vlad Sadovsky), Байрон Шангюон (Byron Changuiion), Купер

- Партин (Cooper Partin), И-Жу Ву (E-Zu Wu), Джим Боуви (Jim Bovee), Джон Фелкинс (John Felkins) и Блэйк Мандерс (Blake Manders);
- ◆ команда Windows SideShow — Дэн Поливи (Dan Polivy);
  - ◆ команда Универсальной аудиоархитектуры (Universal Audio Architecture) Windows (аудио высокого разрешения (High Definition Audio)) — Хакон Странди (Hakon Strande), Франк Беррет (Berrett) и Ченг-мин Лию (Cheng-mean Liu);
  - ◆ команда Windows Virtualization — Джейк Ошинс (Jake Oshins) и Бенджамин Лайс (Benjamin Leis);
  - ◆ команда WinUSB — Рэнди Ол (Randy Aull);
  - ◆ команда Xbox 360 Controller — Мэт Коил (Matt Coill).

*От авторов книги, Пенни Орвик (Penny Orwick),  
Гая Смита (Guy Smith), Кэрол Бухмиллер (Carol Buchmiller),  
Энни Пирсон (Annie Pearson), Гвена Хайба (Gwen Heib),  
и команды Windows Hardware Developer Central (WHDC)  
корпорации Microsoft.*

# **ЧАСТЬ I**

## **Начало работы с моделью WDF**

**Глава 1.** Введение в WDF

**Глава 2.** Основы драйверов под Windows

**Глава 3.** Основы WDF

# ГЛАВА 1

## Введение в WDF

В течение многих лет разработка драйверов для операционных систем Microsoft Windows являлась для разработчиков программного обеспечения трудной задачей. Сложность обучения модели Windows Driver Model (WDM) до недавнего времени ограничивала разработку драйверов сравнительно небольшой группой разработчиков, специализирующихся в этой области.

Модель драйверов Windows Driver Foundation (WDF) более легко изучить, и она облегчает реализацию надежных драйверов для Windows. Модель WDF в значительной степени заменяет модель Windows Driver Model (WDM), и была создана, чтобы позволить разработчикам фокусироваться на требованиях аппаратной части, а не на сложностях операционной системы. Модель WDF также улучшает стабильность системы, позволяя создавать для нескольких важных категорий устройств, для которых раньше требовалось драйверы режима ядра, драйверы, исполняемые в пользовательском режиме.

С помощью модели WDF разработчик может быстро создать простой, но работоспособный драйвер, при этом большая часть обработки событий выполняется механизмом модели WDF. Потом разработчик может постепенно расширять область обрабатываемых событий до тех пор, пока драйвер не будет полностью готов.

Эта книга предназначена для представления модели WDF любому, кто заинтересован в разработке драйверов Windows, включая программистов, не имеющих предыдущего опыта разработки драйверов. Книга была написана в сотрудничестве с командой разработчиков модели WDF компании Microsoft, которые спроектировали ее архитектуру, создали инфраструктуры и разработали образцы драйверов для использования программистами в качестве примеров. Хотя книга начинается с обсуждения на высоком уровне архитектуры и модели программирования WDF, большая ее часть является практическим, основанным на примерах, введением в инфраструктуры WDF для разработки драйверов Windows.

### ***Критерии, примененные при создании интерфейсов API для WDF***

При разработке архитектуры WDF основными и постоянно использующимися критериями, которые мы применяли для добавления интерфейса API, были его характеристики, а именно, является ли данный интерфейс интерфейсом из "тонкой книги" или из "толстой книги".

Интерфейс из "тонкой книги" — это простой интерфейс API, который применялся бы большинством разработчиков драйверов, и чье назначение можно было бы легко определить только по одному его названию (например, такому как `WdfDeviceInitSetExclusive`). В то же самое время, его можно было бы описать в документе небольшого объема, наподобие ко-

роткого доклада для ознакомления клиентов с новым продуктом<sup>1</sup> (отсюда и название — "тонкая книга").

Интерфейс же из "толстой книги" — это трудный для понимания или редко применяемый интерфейс API. Чтобы научиться его применять, разработчику требуется перелопатить и глубоко вникнуть в горы справочной документации.

Этот критерий был первостепенной основой для наших решений относительно того, какие типы интерфейсов API мы предоставили ("тонкая книга") и какую работу оставили выполнять "под капотом" для драйвера инфраструктурам. Мы надеемся, что принятые нами решения были правильными, и вы получите удовольствие от подробного изучения инфраструктур по этой "толстой книге". (Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Об этой книге

Эта книга организована таким образом, чтобы предоставить основную информацию, необходимую для разработки драйверов WDF; предполагается, что у читателя нет предварительного опыта разработки драйверов. Книга начинается с описания работы драйвера в среде операционной системы Windows, а затем на основе этой информации описывается применение модели WDF для создания драйверов.

## Кому адресована эта книга

Эта книга предназначена для разработчиков, которые умеют программировать на языке C или C++ и хотят создавать драйверы Windows, включая:

- ◆ **разработчиков драйверов, которые хотят научиться создавать драйверы с помощью модели WDF.** Опытным разработчикам драйверов не составит труда адаптироваться к новой модели. Но драйверы WDF пользователяского режима основаны на СОМ-технологии, поэтому в книге дается введение в основы программирования СОМ для тех, кто незнаком с этой технологией;
- ◆ **разработчиков приложений, которые хотят начать разрабатывать драйверы.** Модель WDF намного легче в изучении, чем более ранние драйверные модели, но приложения во многом отличаются от драйверов, особенно от драйверов режима ядра. В книге представлена основная вводная информация, чтобы помочь вам понять структуру и функционирование драйверов Windows. Эта вводная информация потом применяется при рассмотрении разработки драйверов на протяжении всей книги;
- ◆ **разработчиков аппаратного обеспечения, которым необходимо понимать, каким образом драйверы взаимодействуют с устройствами.** Разработчики аппаратного обеспечения, которым нужно создавать драйверы для прототипов своей аппаратуры, найдут модель WDF особенно полезной, т. к. она позволяет быстро создавать прототипы драйверов и более легкая в изучении.

Разработка драйверов с использованием модели WDF требует знания языка программирования C++, а драйверы режима ядра почти всегда пишутся на языке C. Если вы не знаете ни одного из этих языков программирования, вам следует пользоваться для справки какой-либо из многочисленных книг по этим языкам. Используя ваши существующие знания языков C и

---

<sup>1</sup> По-английски *white paper* — в данном случае короткий документ для ознакомления потребителей с возможностями нового продукта. — Пер.

C++ и программирования под Windows как основу, данная книга предоставляет понятия, руководящие принципы, примеры программирования и подсказки, чтобы позволить вам начать разработку драйверов с помощью модели WDF.

## Часть I. Начало работы с моделью WDF

В части I книги представляются инструменты и ресурсы, основные понятия драйверов и операционной системы Windows и обзор модели WDF.

**Глава 1. Введение в WDF.** В этой главедается общее представление о книге и инструментах разработки, описываемых в ней. Начните со следующего:

- ◆ просмотрите эту главу, чтобы получить представление о структуре книги и соглашениях, применяемых в тексте и образцах кода;
- ◆ ознакомьтесь с системными требованиями и установите набор Windows Driver Kit (WDK) и отладочные инструменты согласно руководству, изложенному в разд. "Приступая к разработке драйверов" далее в этой главе, чтобы можно было исследовать и попробовать выполнить примеры кода.

**Глава 2. Основы драйверов под Windows.** В этой главе изложен вводный материал, необходимый для понимания процесса разработки драйверов. Особенно полезной будет эта глава для программистов приложений, не имеющих опыта программирования драйверов или опыта программирования в режиме ядра.

В ней вы сможете быстро получить основную информацию об архитектуре Windows, модели ввода/вывода Windows, различии между пользовательским режимом и режимом ядра, технологии Plug and Play, прерываниях, управлении памятью, потоках и синхронизации, а также о других ключевых понятиях.

Вопросы, затронутые в главе 2, рассматриваются более подробно далее в книге, при изучении разработки драйверов модели WDF.

**Глава 3. Основы WDF.** Модель WDF состоит из трех основных составляющих: инфраструктуры UMDF (User-Mode Driver Framework, инфраструктура драйвера пользовательского режима), инфраструктуры KMDF ((Kernel-Mode Driver Framework, инфраструктура драйвера режима ядра) и набора инструментов для тестирования и верификации. В этой главе дается концептуальное обозрение архитектуры WDF и представляются эти два вида инфраструктур (в дальнейшем называемые *драйверные инфраструктуры* или просто *инфраструктуры*). Также представляются основные понятия, связанные с моделью WDF, такие как объектная модель WDF, модель ввода/вывода, и каким образом WDF управляет событиями Plug and Play и событиями энергопотребления.

## Часть II. Изучение инфраструктур

Хотя оба типа инфраструктур WDF во многом сходны, они не идентичны, и каждая имеет свои достоинства и ограничения. В части II представляется детальное обозрение инфраструктур.

**Глава 4. Обзор драйверных инфраструктур.** В этой главе описываются два типа инфраструктур, включая их связанные компоненты среды исполнения:

- ◆ инфраструктура UMDF позволяет создавать простые, надежные драйверы Plug and Play для нескольких категорий устройств, особенно для устройств, ориентированных на потребителя, таких как портативные медиапроигрыватели и сотовые телефоны;

- ◆ инфраструктура KMDF позволяет создавать драйверы, которые должны исполняться в режиме ядра, т. к. они осуществляют прямой доступ к памяти (DMA), обрабатывают прерывания, используют циклы с жесткими временными требованиями, являются клиентами других драйверов режима ядра или требуют ресурсов режима ядра, таких как, например, нестаничный пул.

В главе также предоставляются рекомендации по выбору типа инфраструктуры для создания драйвера для определенного устройства.

**Глава 5. Объектная модель WDF.** Инфраструктура WDF поддерживает объектно-ориентированную событийно-управляемую модель программирования, в которой:

- ◆ основными строительными блоками драйверов являются объекты. Объектные модели инфраструктур UMDF и KMDF похожи друг на друга, но отличаются в деталях реализации;
- ◆ с каждым объектом ассоциируется набор событий. Обработка по умолчанию для большинства событий предоставляется инфраструктурами. Драйверы обрабатывают только те события, которые имеют отношение к их устройству, и оставляют обработку других событий инфраструктурам.

Драйвер WDF взаимодействует с этими объектами посредством постоянных, четко определенных интерфейсов программирования. В этой главе предоставлены подробности объектной модели в качестве основы для понимания, какие функциональности нужно реализовать в драйвере, а какие оставляются для исполнения драйверным инфраструктурам.

**Глава 6. Структура драйвера и его инициализация.** С целью помочь вам начать разбираться в особенностях разработки драйверов Windows, в этой главе исследуется структура и обязательные компоненты драйверов WDF. Также рассматриваются общие аспекты драйверов UMDF и KMDF: объекты драйверов, объекты устройств, точки входа драйверов и callback-функции. Кроме этого, изучаются входные процедуры, инициализация и создание объектов устройств.

## Часть III. Применение основ WDF

Когда вы уверенно разбираетесь в архитектуре и компонентах обеих инфраструктур WDF, можно приняться за изучение деталей драйверов Windows, которым вам придется уделять большую часть времени и усилий при разработке. В этой части книги исследуется несколько важных понятий и практикуемых процедур.

**Глава 7. Plug and Play и управление энергопотреблением.** Технология Plug and Play представляет собой комбинацию аппаратного и программного обеспечения, позволяющую компьютеру распознавать и поддерживать изменения в аппаратной конфигурации при минимальном или отсутствующем участии пользователя. Windows также поддерживает архитектуру управления энергопотреблением, которая предоставляет всестороннюю политику для управления энергопотреблением компьютерной системы и подсоединенных к ней устройств.

Работать с этими двумя возможностями Windows оказалось трудной задачей для разработчиков драйверов, использующих предыдущие драйверные модели. В этой главе исследуется, каким образом эти две возможности работают в драйверах WDF, и показывается, каким образом драйверные инфраструктуры значительным образом сокращают и упрощают код, необходимый для поддержки механизмов Plug and Play и управления энергопотреблением.

**Глава 8. Поток и диспетчеризация ввода/вывода.** Драйвер Windows получает от приложений запросы ввода/вывода, обслуживает их, после чего возвращает информацию приложению. В этой главе дается общее описание хода ввода/вывода в драйверах WDF и типы ввода/вывода, запросы на обслуживание которых драйвер может получить, и способы создания очередей для обработки этих запросов. Особое внимание, с вниканием в подробности, уделяется некоторым часто используемым типам запросов.

**Глава 9. Получатели ввода/вывода.** Хотя драйверы могут полностью выполнять некоторые запросы ввода/вывода, может быть необходимым передать им другие запросы на исполнение более низким компонентам своего стека устройств или вообще другим стекам устройств. Драйверы также могут сами давать запросы ввода/вывода. Получатель ввода/вывода применяется драйвером WDF для передачи запроса ввода/вывода другому драйверу, независимо от того, находится ли этот другой драйвер в том же самом или другом стеке устройств. В этой главе подробно рассматривается, каким образом создаются получатели ввода/вывода и посылаются запросы ввода/вывода, включая информацию о специализированных целях для устройств USB.

**Глава 10. Синхронизация.** Windows является операционной системой с вытесняющей многозадачностью. Это означает, что разные потоки могут пытаться получить одновременный доступ к разделяемым структурам данных или ресурсам, и что в одно и то же время (параллельно) может исполняться несколько драйверных процедур. Чтобы обеспечить сохранность данных и предотвратить состояния гонок, все драйверы должны согласовывать во времени или, другими словами, синхронизировать доступ к разделяемым структурам данных и ресурсам. В этой главе обсуждаются ситуации, требующие синхронизации; после этого рассматриваются функции синхронизации и параллелизма, предоставляемые драйверными инфраструктурами.

**Глава 11. Трассировка и диагностируемость драйверов.** Трассировка программы предоставляет метод анализа поведения драйверов, не требующий высоких накладных расходов. В этой главе рассматривается использование Windows Software Trace Preprocessor (WPP) для оборудования драйвера WDF средствами, с помощью которых можно анализировать поведение драйвера и устранить возможные проблемы в нем. Большое внимание в этой главе уделяется вопросу применения самых лучших приемов для создания драйверов, легко поддающихся диагностике.

**Глава 12. Вспомогательные объекты WDF.** Во всех драйверах применяются объекты устройств, драйверы и очереди ввода/вывода, описанные в предыдущих главах. Драйверные инфраструктуры также определяют дополнительные объекты, которые представляют менее распространенные абстракции. В этой главе описываются некоторые из этих других объектов, которые вам придется использовать при создании драйверов WDF. Также рассматриваются методы выделения памяти, чтения и записи в реестр, использование таймеров и коллекций и поддержка Windows Management Instrumentation (WMI, инструментарий управления Windows) в драйверах KMDF.

**Глава 13. Шаблон UMDF-драйвера.** Образец драйвера *skeleton* (каркас) содержит минимальный объем кода, необходимый для драйвера UMDF. Его можно использовать в качестве отправной точки для создания драйверов для настоящего аппаратного обеспечения. В этой главе сначала объясняется, каким образом в драйвере *skeleton* демонстрируются минимальные требуемые возможности и оптимальные методики драйвера UMDF. После этого рассматривается, каким образом данный образец можно использовать как основу для создания полноценного драйвера.

## Часть IV. Смотрим глубже – больше о драйверах WDF

В сущности, драйверы режима ядра являются частью операционной системы Windows и, соответственно, должны обрабатывать осложнения, с которыми драйверам пользовательского режима не приходится сталкиваться. Драйверам KMDF, может быть, необходимо иметь дело с тонкостями аппаратных прерываний и прямого доступа к памяти. Для драйверов UMDF вам необходимо понимать, как использовать и создавать объекты COM. Эти более глубокие темы и исследуются в данной части книги.

**Глава 14. За пределами инфраструктуры.** Хотя инфраструктуры предоставляют большинство возможностей, используемых драйверами в большинстве случаев, иногда драйверу требуются сервисы, не поддерживаемые инфраструктурами. Использование таких системных сервисов, не предоставляемых инфраструктурой, описывается в этой главе. Например:

- ◆ драйверы UMDF могут использовать многие функции Windows API;
- ◆ драйверы KMDF могут использовать системные функции режима ядра, включая функции, которые манипулируют объектами WDM, лежащими в основе объектов WDF.

**Глава 15. Диспетчеризация, контекст потока и уровни IRQL.** На работу драйверов режима ядра влияют такие факторы, как диспетчеризация потоков, контекст потока и текущий уровень прерывания (IRQL) для каждого процессора. В этой главе исследуются понятия и оптимальные методики, которыми вы должны владеть, чтобы избежать в драйверах KMDF проблем, связанных с прерываниями, вытеснением потоков и IRQL.

**Глава 16. Аппаратные ресурсы и прерывания.** Если аппаратное обеспечение устройства генерирует прерывания, драйвер режима ядра должен обслуживать эти прерывания. Для обслуживания аппаратного прерывания драйвер KMDF должен создать объект прерывания, включить и выключить прерывание и отреагировать должным образом на возникшее прерывание. В этой главе обсуждается обслуживание прерываний с помощью инфраструктуры KMDF и предоставляются соответствующие рекомендации и оптимальные методики.

**Глава 17. Прямой доступ к памяти.** DMA (Direct Memory Access, прямой доступ к памяти) — это высокопроизводительный способ обмена данными непосредственно с памятью, минуя процессор. DMA поддерживает более высокую скорость передачи данных, чем другие подходы, при более низком общесистемном расходовании процессорного времени. Большая часть работы по реализации в драйвере функциональности DMA выполняется прозрачно инфраструктурой KMDF. В этой главе излагаются основные понятия механизма DMA и способы его реализации в драйверах KMDF.

**Глава 18. Введение в COM.** Для создания драйвера UMDF необходимо использовать несколько объектов COM (Component Object Model, модель составных объектов), которые принадлежат к среде исполнения UMDF. Кроме этого, необходимо создать несколько callback-объектов на основе COM. В этой главе дается введение в основы использования и создания COM-объектов для нужд UMDF.

## Часть V. Создание, установка и тестирование драйверов WDF

Для создания, тестирования, отладки и установки драйверов применяется набор инструментов и методик, созданных специально для разработки драйверов. Кроме стандартных инструментов, WDF содержит дополнительный набор инструментов для верификации, тестирования и отладки, которые облегчают создание надежных драйверов WDF.

**Глава 19. Сборка драйверов WDF.** Драйверы для Windows создаются с помощью утилиты сборки Build.exe из набора WDK. Эта утилита командной строки применяется компанией Microsoft для сборки самой Windows. В этой главе объясняется настройка среды сборки и процесс сборки драйверов UMDF и KMDF.

**Глава 20. Установка драйверов WDF.** Установка драйверов значительно отличается от установки обычных приложений. В этой главе исследуются инструменты и процессы для установки драйверов, включая такие инструменты, как DevCon и Device Manager. Также тщательно рассматриваются проблемы с INF-файлами для драйверов WDF и предоставляются ссылки на ресурсы для создания цифровой подписи драйверов.

**Глава 21. Инструменты для тестирования драйверов WDF.** Всеобъемлющее тестирование на всех стадиях разработки является абсолютно необходимым для создания надежного, высококачественного драйвера. В дополнение к Driver Verifier и другим средствам тестирования драйверов общего назначения, предоставленных в наборе WDK, WDF содержит встроенное средство динамической верификации драйверов для обеих инфраструктур. В этой главе дается краткое обозрение инструментов и оптимальных методик для тестирования и верификации драйверов WDF.

**Глава 22. Отладка драйверов WDF.** Наиболее предпочтительным отладчиком как для UMDF-, так и для KMDF-драйверов является WinDbg. Этот отладчик может работать как в режиме ядра, так и в пользовательском режиме, и поддерживает несколько расширений, которые упрощают отладку проблем, специфичных для WDF. В этой главе дается введение в отладку драйверов и рассматриваются основы использования отладчика WinDbg для отладки драйверов WDF.

**Глава 23. Инструмент PREfast for Drivers.** Инструмент PREfast for Drivers — это инструмент статической проверки исходного кода программных драйверов под Windows. Этот инструмент применяется при компиляции драйвера и предоставляет отчет о различных типах ошибок в коде драйвера, которые не могут быть выявлены компилятором и при исполнении. В этой главе рассматривается, каким образом инструмент PREfast можно эффективно применить при разработке драйверов, включая использование примечаний в исходном коде драйвера, чтобы PREfast мог выполнить более глубокий анализ кода.

**Глава 24. Инструмент Static Driver Verifier.** Инструмент Static Driver Verifier (SDV) — это средство статической верификации драйверов режима ядра, которое эмулирует прохождение операционной системой через код драйвера и условно исполняет исходный код. Этот инструмент содержит модель операционной системы Windows и набор правил использования драйверами интерфейсов Windows. В главе описывается применение SDV в качестве рекомендуемой оптимальной методики при разработке драйверов. Также предоставлены детали о правилах KMDF для SDV.

## Словарь

Словарь содержит список терминов и сокращений, использованных в этой книге. Всеохватывающий глоссарий терминов, применяющихся в области разработки драйверов, имеется в документации WDK.

## Условные обозначения

Большинство условных обозначений, применяемых в этой книге, такие же, как и соответствующие условные обозначения в документации по WDK и другой справочной документа-

ции компании Microsoft для программистов. Кроме этого, в книге предоставлены ссылки, чтобы помочь вам найти образцы, документацию, доклады, инструменты и другие ресурсы по каждому рассматриваемому предмету. Типографские и другие условные обозначения, использованные в этой книге, приведены в табл. 1.1.

**Таблица 1.1. Условные обозначения**

Условное обозначение	Описание
<b>Полужирный шрифт</b>	Элементы интерфейса программ, интернет-адреса, а также фрагменты текста, на которые следует обратить внимание
<b>Курсив</b>	Определения и фрагменты текста, на которые следует обратить внимание
Моноширинный	Названия предоставляемых системой или определенных в системе функций и вспомогательных процедур, членов структур, перечислений и ключей реестра. Например, <code>WdfCreateDevice</code> обозначает предоставляемую системой функцию, поддерживающую драйверы WDF режима ядра. Моноширинным шрифтом также выделены примеры кода, например: <code>hwInitData.DeviceIdLength=4</code>
Моноширинный курсив	Заполнитель для имен функций, формальных параметров или любого другого текста, который заменяется необходимым именем в готовом коде. Части путей реестра или элементы INF-файлов, выделенные курсивом, являются заполнителями, которые нужно заменить конкретными текстовыми значениями для данного драйвера или устройства. Например: <ul style="list-style-type: none"> <li>• <code>EvtDriverDeviceAdd</code> — это заполнитель для имени функции обратного вызова, которая определяется драйвером;</li> <li>• в предупреждении <code>#pragma warning (disable:WarningNumber)</code> часть <code>WarningNumber</code> служит заполнителем для числового значения, которое будет выведено в настоящем предупреждении</li> </ul>
<b>ВЕРХНИЙ РЕГИСТР</b>	Идентификаторы констант, имена типов данных, побитовые операторы и предоставляемые системой макросы. Идентификаторы в верхнем регистре необходимо вводить точно так, как они показаны.  Например, идентификатор <code>WDF_DRIVER_CONFIG</code> является системной структурой
Filename.txt	Имя файла. В этой книге имена файлов записываются в верхнем и нижнем регистре, чтобы их было легче читать. При этом имена файлов не чувствительны к регистру
%wdk%	Корневой установочный каталог для WDK; обычно это будет <code>C:\Win\DDK\НомерСборки</code>
%windir%	Корневой установочный каталог Windows; обычно это будет <code>C:\Windows</code>
x86, x64, Itanium	Ссылки на различные архитектуры центрального процессора, исполняющие Windows, а именно: <ul style="list-style-type: none"> <li>• x86 для 32-битных процессоров, исполняющих набор команд Intel;</li> <li>• x64 для 64-битных процессоров, таких как, например, AMD64;</li> <li>• Itanium для процессоров Itanium компании Intel</li> </ul>
\<i386   amd64   ia64>	Альтернативные подпапки в папке WDK, содержащие файлы для различных аппаратных платформ, а именно: <ul style="list-style-type: none"> <li>• i386 для x86-версий Windows;</li> <li>• amd64 для x64-версий Windows;</li> <li>• ia64 для 64-битных версий Windows на платформе Itanium.</li> </ul> Например: <code>%wdk%\tools\acpi\i386\</code>

### **Расположение ресурсов, требуемых для каждой главы**

Каждая глава начинается со списка примеров, документации и инструментов, необходимых для выполнения рассматриваемого в ней материала на вашей системе. Далее показан пример такого списка:

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b> Build.exe	%wdk%\bin\<amd64   ia64   i386>
<b>Образцы драйверов</b> Fx2_Driver Osrusbfx2	%wdk%\src\umdf\usb\fx_2driver%wdk%\src\kmdf\Osrusbfx2
<b>Документация WDK</b> Объекты и интерфейсы UMDF	<a href="http://go.microsoft.com/fwlink/?LinkId=79583">http://go.microsoft.com/fwlink/?LinkId=79583</a>
<b>Прочее</b> "Разработка драйверов с помощью WDF: новости и сводки" на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=80911">http://go.microsoft.com/fwlink/?LinkId=80911</a>

Темы WDK, перечисленные в списке ресурсов для каждой главы, включают онлайновые версии документации MSDN. Другие ссылки указывают на различные доклады и интернет-ресурсы. Ссылки на все эти и другие ресурсы можно также найти на Web-странице "Developing Drivers with WDF: News and Updates" ("Разработка драйверов с помощью WDF: Новости и последняя информация") Web-сайта WDHC (Windows Hardware Developer Central, Центр разработчиков аппаратного обеспечения Windows).

## **Приступая к разработке драйверов**

Чтобы повторить примеры, приведенные в книге, и использовать инфраструктуру WDF, необходимо установить на ваш компьютер текущую версию набора Windows Driver Kit (WDK, набор разработки драйверов Windows). Набор WDK содержит большинство ресурсов, необходимых для разработки драйверов, такие как инструменты, документация и библиотеки. В следующих разделах даются руководящие указания и советы по установке программного обеспечения и нахождению образцов и инструментов, рассматриваемых в книге.

### **Внимание!**

Всегда пользуйтесь самой последней версией набора WDK. Книга предполагает, что вы пользуетесь версией Build 6000 или более поздней версией WDK. Если вы уже занимаетесь разработкой драйверов Windows, чтобы иметь компоненты, рассматриваемые в книге, вам необходимо установить Build 6000 или более новую версию WDK. Инструмент Static Driver Verifier для KMDF-драйверов и примечания для конкретных драйверов в инструменте PREfast требуют версию WDK, поставляемую с выпуском Beta 3 Windows Microsoft Server с кодовым названием Longhorn, или более новую версию WDK.

## **Системные требования для разработки драйверов**

Драйверы KMDF можно разрабатывать и создавать для Windows 2000 или более новых версий Windows. Драйверы UMDF можно разрабатывать и создавать для Windows XP или бо-

лее новых версий Windows. Для сборки драйверов можно применять любую последнюю версию Windows. Чтобы настроить драйвер на конкретную версию и архитектуру центрального процессора, при запуске утилиты сборки Build указывается соответствующая конфигурация среды.

Но вы должны планировать установку, тестирование и отладку разрабатываемого драйвера на системе под управлением целевой версии Windows и с аппаратной частью такой же, как аппаратная часть системы заказчика, для которой предназначается разрабатываемый драйвер, или очень близкой к ней.

### Внимание!

Всегда пользуйтесь наиболее поздней версией пакета отладочных инструментов Debugging Tools for Windows. Как приобрести этот пакет, объясняется далее в этой главе.

Для отладки драйверов KMDF требуются два компьютера: один для исполнения отладчика и второй для исполнения драйвера, подлежащего отладке. Ошибки драйверов режима ядра обычно вызывают системные сбои, в результате чего может быть нарушена целостность файловой системы, что, в свою очередь, может вызвать потерю данных. Поэтому необходимо разнести отладчик и драйвер на разные компьютеры.

Для отладки драйверов UMDF отладчик и драйвер могут исполняться или на одной и той же, или же на отдельных системах. Многие из этих задач можно также выполнять на компьютере с разными версиями Windows, установленными на разных разделах диска. Но общепринятой практикой является использование одного компьютера для разработки и, по крайней мере, одного другого компьютера, выделенного исключительно для тестирования и отладки.

Мы рекомендуем следующую конфигурацию аппаратной части компьютера для разработки.

- ◆ **Компьютер с многоядерным процессором или с несколькими процессорами.** Как минимум, процессор должен поддерживать гиперпотоковость (hyperthreading). При тестировании драйверов только на однопроцессорной системе некоторые виды ошибок, такие как, например, состояние гонок, могут быть невыявлены. Кроме этого, согласно требованиям программы Windows Logo Program, все драйверы, представленные на внесение в эту программу, должны пройти тестирование на работу в многопроцессорных системах.
- ◆ **Компьютер с 64-битным процессором под управлением x64-версии Windows.** Некоторые типы критических ошибок можно выявить только на 64-битных системах. Согласно требованиям программы Windows Logo Program, драйвер должен поддерживать как 32-битные, так и 64-битные системы.

### Подсказка

Если вы разрабатываете и тестируете драйверы WDM с помощью набора WDK версии Build 6000 или более поздней, вы также можете использовать эти среды для разработки драйверов с WDF.

## Как приобрести и установить набор WDK

Набор WDK, в который входит набор Driver Development Kit (DDK, набор разработки драйверов), является основным ресурсом разработчика драйверов. Набор WDK содержит большую часть средств, необходимых для разработки драйверов, которые описаны в следующих разделах.

Набор WDK поддерживает разработку драйверов для семейства операционных систем Microsoft Windows NT, начиная с Windows 2000. Он выпускается периодически, при этом версия конкретного набора обычно соотносится с определенным выпуском Windows. Но это делается всего лишь для удобства указания даты выпуска. В действительности, каждый выпуск набора WDK поддерживает создание драйверов для всех аппаратных платформ и всех версий Windows, которые компания Microsoft поддерживает согласно своей политике предоставления поддержки своим продуктам на протяжении их жизненного цикла. Набор WDK выпуска 2007 года поддерживает создание драйверов для Windows 2000, Windows XP, Windows Server 2003, Windows Vista и Windows Server Longhorn.

Всегда используйте самую последнюю версию набора WDK. Следование этой практике гарантирует, что у вас всегда будет текущая документация и инструменты, включая все обновления и исправления, выпущенные после предыдущей версии набора.

### Как приобрести набор WDK:

1. За информацией, как приобрести текущую версию набора WDK, обратитесь к Web-сайту WHDC по адресу <http://www.microsoft.com/whdc/>.

Новые версии повляются совместно с выпусками связанных продуктов, например, бета и RTM-выпусками<sup>1</sup> Windows, или важными событиями в сфере разработки драйверов, такими как, например, WinHEC (Windows Hardware Engineering Conference, конференция по разработке аппаратного обеспечения под Windows). Информацию о том, какие версии Windows поддерживаются текущим набором WDK, можно также найти на Web-сайте WHDC.

2. Если вы скачаете образ ISO с WDK, создайте установочный диск, записав образ на CD или DVD.

### Как установить набор WDK:

1. Вставьте установочный диск WDK в привод и запустите программу установки.
2. Прочтите информацию по выпуску (Release Notes) WDK, чтобы убедиться, что нет никаких вопросов, которые могут повлиять на установку.
3. Установите необходимые компоненты, обязательные перед установкой набора WDK.
4. Программа установки проверяет систему на наличие требуемых компонентов и включает кнопки в разделе **WDK Prerequisite Setup Packages**<sup>2</sup> диалогового окна установки напротив компонентов, требуемых для работы набора WDK, но не установленных в системе. Возможно, что вам не будут нужны все эти компоненты, но, скорей всего, вам нужна будет вторая версия .NET Framework, которая требуется практически для всех современных приложений, а также Microsoft Document Explorer для просмотра документации.
5. Внизу раздела **WDK Setup Package Features** окна установки нажмите кнопку **Install**.
6. По умолчанию, WDK устанавливается на диск C: в корневой каталог установки **WinDDK\НомерСборки**. При установке нескольких версий WDK на один компьютер мастер установки WDK помещает каждую версию в свой отдельный каталог.

<sup>1</sup> Release to Manufacturing — окончательный выпуск продукта для его промышленного производства, в отличие от бета или подобных выпусков. — *Пер.*

<sup>2</sup> Программные пакеты, которые должны быть установлены, перед тем как может быть установлен набор WDK. — *Пер.*

### Подсказка

Так как WDK можно установить на другие диски кроме C:, для обозначения корневого каталога установки в этой книге используется переменная среды %wdk% .

## Библиотеки WDK

Набор WDK содержит статические библиотеки (LIB), динамические библиотеки (DLL) и библиотеки WDF, необходимые для разработки драйверов.

- ◆ **Библиотеки DLL соинсталляторов.** Набор WDK содержит соинсталляторы<sup>1</sup> для инфраструктур UMDF и KMDF, которые разработчик может распространять со своими продуктами. При установке драйверов соинсталляторы устанавливают средства поддержки среды исполнения соответствующей инфраструктуры на компьютер пользователя, если среда исполнения еще не была установлена. Драйверы динамически связываются с инфраструктурами, установленными соинсталляторами. Для облегчения распознавания соинсталляторов их имена содержат номер версии WDF. Для каждой поддерживаемой архитектуры центрального процессора имеется собственная версия соинсталлятора.

Подробное описание соинсталляторов приводится в *главе 20*.

- ◆ **Расширения отладчика.** Расширения отладчика WDF — это команды, специализированные под WDF, которые исполняются в контексте отладчика WinDbg. Расширения отладчика упакованы в две библиотеки DLL: файл WudfExt.dll содержит расширения UMDF, а файл WdfKd.dll — расширения KMDF. Для каждой поддерживаемой архитектуре центрального процессора имеются собственные версии расширений отладчика.

Отладчик WinDbg и его расширения обсуждаются в *главе 22*.

- ◆ **Библиотеки.** Набор WDK содержит несколько статических библиотек. Драйверы KMDF статически связываются с библиотеками WdfDriverEntry.lib и WdfLdr.lib. Драйверы UMDF можно создавать с помощью библиотеки Active Template Library (ATL). Эта библиотека шаблонов на языке C++ создана, чтобы упростить процесс разработки объектов COM.

Файлы библиотеки KMDF находятся в папке %wdk%\lib\wdf\kmdf, а библиотеки ATL — в папке %wdk%\lib\atl.

## Документация WDK

В этой книге дается всего лишь введение в разработку драйверов с помощью WDF. А чтобы получить все информацию, необходимую для разработки полноценного драйвера устройства, вам нужно будет обратиться к документации по WDK. В этой документации изложена подробная справочная информация для каждой функции, предоставленной в интерфейсе драйверов WDF-устройства. Документация WDK также содержит концептуальный материал, руководства по разработке и реализации драйверов и документацию для инструментов WDK.

**Чтобы запустить документацию WDK с помощью меню Пуск, нажмите кнопку Start (Пуск) на панели задач, после чего выберите последовательность элементов, вложенных**

---

<sup>1</sup> Соинсталлятор выполняет операции установки, которые специфичны для конкретного устройства. — Пер.

меню: **All Programs | Windows Driver Kits | НомерСборки | Help | WDK Documentation** (Программы | Windows Driver Kits | НомерСборки | Help | WDK Documentation).

Документацию WDK можно также просмотреть в библиотеке MSDN в Интернете по адресу <http://msdn.microsoft.com>. Она находится в статье "Windows Driver Kit" в разделе **Microsoft Win32 and COM Development** таблицы содержания библиотеки MSDN. Онлайновая версия документации WDK обновляется ежеквартально.

## Инструменты WDK

Набор WDK содержит значительное количество инструментов для разработки и тестирования драйверов. Большинство из них являются инструментами командой строки и исполняются в командном окне, но есть традиционные приложения Windows с графическим интерфейсом пользователя. Некоторые инструменты для тестирования перечислены далее.

- ◆ **Инструменты для трассировки.** В главе 11 описывается инструмент WPP и связанные инструменты WDK для трассирования исполнения и отладки драйверов.
- ◆ **Driver Verifier и другие инструменты для динамического тестирования.** В главе 21 рассматриваются различные инструменты для тестирования драйверов.
- ◆ **Статические верификаторы.** В главах 23 и 24 детально описывается использование инструментов статической верификации.

### Примечание

Чтобы использовать некоторые из этих инструментов в Windows Vista, необходимо указать, что данная утилита должна исполняться с повышенными привилегиями, даже если вы уже и вошли в систему, как пользователь с повышенными привилегиями. Для подробной информации на эту тему, см. статью "User Account Control" ("Управления учетной записью пользователя") в онлайновой версии MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=80621>.

**Чтобы запустить приложение с повышенными привилегиями в Windows Vista:**

1. Нажмите кнопку **Start** (Пуск) на панели задач, щелкните правой кнопкой мыши по необходимому приложению, после чего выберите команду **Run as administrator**.
2. Если вы уже вошли в систему с правами администратора, Windows Vista выведет диалоговое окно **User Account Control**, запрашивая разрешение продолжить исполнение.
3. Для запуска приложения нажмите кнопку **Continue**.
4. Если же у вас нет прав администратора, Windows Vista потребует предоставить параметры доступа администратора.

**Чтобы открыть командное окно с повышенными привилегиями:**

1. Нажмите кнопку **Start** (Пуск) на панели задач, щелкните правой кнопкой мыши по приложению **Command Window**, после чего выберите команду **Run as administrator**.
2. Нажмите кнопку **Continue** и предоставьте параметры доступа администратора, если система их потребует.

Любое приложение, исполняемое в этом командном окне, в том числе любое приложение Windows, также будет иметь повышенные привилегии.

## Образцы WDK

Набор WDK содержит большое количество образцов распространенных типов драйверов. Эти образцы содержат хорошо продуманный рабочий код и снабжены многочисленными комментариями.

Образцы WDF устанавливаются при инсталляции WDK в папки %wdk%\src\kmdf и %wdk%\src\umdf. Каждый образец устанавливается в свою отдельную папку, с таким же именем, как и имя образца драйвера.

### Подсказка

Прежде чем компилировать или модифицировать образец драйвера WDK, скопируйте файлы данного образца в другую папку и впоследствии работайте с этими копиями. Таким образом, оригинальные образцы будут сохранены для дальнейшего использования.

## Образцы UMDF

Если вы никогда раньше не работали с WDF, рекомендуется сначала изучить образцы Skeleton и Fx2\_Driver, которые используются на протяжении всей книги.

- ◆ **Образец драйвера Skeleton (Шаблон UMDF).** Этот базовый драйвер не исполняет никаких операций, кроме как загрузка и запуск на исполнение.

В данном случае ничего большего и не требуется, т. к. образец драйвера Skeleton предназначен для изучения основ работы драйверов UMDF, и его простота способствует более легкому пониманию этих принципов. Но этот образец можно также использовать как начальную заготовку для разработки полноценного драйвера на его основе, т. к. в большинстве драйверов UMDF используется большая часть кода в этом образце, не требующая никакой или очень незначительной модификации. Код к начальной заготовке можно добавлять пошагово, чтобы обеспечить обработку требований конкретного устройства.

Образец драйвера Skeleton устанавливается при установке набора WDK в папку %wdk%\src\umdf\skeleton.

- ◆ **Драйвер Fx2\_Driver.** Это образец драйвера для USB-устройств, разработанный специально для целей обучения.

Драйвер Fx2\_Driver работает с обучающим устройством OSR USB FX2 Learning Kit, выпускаемым компанией Open System Resources (OSR). Информация об этом устройстве и его использовании приводится далее в этой главе. Простота драйвера Fx2\_Driver обусловлена простотой устройства, для поддержки работы которого он предназначен. Но, несмотря на свою простоту, драйвер работает с настоящим устройством и демонстрирует диапазон основных возможностей UMDF, включая обработку запросов устройства на чтение, запись и ввод/вывод.

Код из драйвера Fx2\_Driver используется на протяжении всей этой книги в примерах, объясняющих работу драйверов UMDF. Другие образцы набора WDK, также содержащиеся в этой книге, демонстрируют возможности, не поддерживаемые драйвером Fx2\_Driver.

Драйвер Fx2\_Driver хранится в папке %wdk%\src\umdf\usb\fx2\_driver, при этом в подпапках этой папки, поименованных от Step1 до Step5, находятся разные версии этого драйвера. В подпапке Step1 находится версия драйвера, реализующая минимальные возможности. По мере возрастания индекса подпапки содержащаяся в ней версия драйвера об-

ладает все более расширенными возможностями. В папке Final находится завершенный драйвер, который также используется в этой книге. В папке \Exe находится исходный код простого консольного приложения для работы с устройством и изучения возможностей драйвера.

## Образцы KMDF

В наборе WDK также присутствуют многочисленные образцы драйверов KMDF. Если вы никогда раньше не работали с набором WDF, рекомендуется сначала изучить образцы Toaster и Osrusbf1x, которые используются на протяжении всей книги. В других образцах демонстрируются возможности, которые не поддерживаются этими двумя драйверами, например DMA и обработка прерываний.

- ◆ **Toaster.** Этот драйвер эмулирует поведение настоящих устройств в Windows.

Драйвер Toaster несколько посложнее, чем драйвер Skeleton, но, тем не менее, он также может быть полезен как отправная точка в изучении драйверов KMDF. Драйвер Toaster составлен из нескольких связанных драйверов, включая драйверы фильтра, шины и функций.

Драйвер Toaster находится в папке %wdk%\src\kmdf\toaster. Подпапки этой папки содержат разные связанные драйверы. Лучше всего будет начать изучение этой группы драйверов с драйвера функции в папке %wdk%\src\kmdf\toaster\func.

В этой папке находятся две разные версии драйвера: базовая, под названием Simple, и полнофункциональная, под названием Featured. В папке %wdk%\src\kmdf\toaster\exe находится исходный код для нескольких приложений, с помощью которых можно исследовать возможности драйвера.

- ◆ **Osrusbf1x2.** Это драйвер режима ядра для устройств USB.

Код из драйвера Osrusbf1x используется на протяжении всей этой книги в примерах, объясняющих работу драйверов KMDF. В этом образце не демонстрируются все существующие возможности KMDF. Папка %wdk%\src\kmdf содержит другие образцы, демонстрирующие возможности, не поддерживаемые драйвером Osrusbf1x.

Драйвер Osrusbf1x — это эквивалент режима ядра драйвера Fx2\_Driver. Его возможности почти такие же, как и драйвера Fx2\_Driver, а структура и код тоже очень сходные.

Драйвер Osrusbf1x находится в папке %wdk%\src\kmdf\osrusbf1x с шестью подпапками. В папках Step1 по Step5 хранятся разные версии драйверов, возрастающие по сложности с увеличением индекса папки. Возможности каждой версии подобны возможностям соответствующей версии драйвера Fx2\_Driver. В папке \Exe находится исходный код для простого приложения тестирования.

### Подсказка

Тестовое приложение Osrusbf1x2 можно использовать как с OsrUsbFx2, так и с Fx2\_Driver драйвером. Приложение позволяет получить доступ ко всем возможностям устройства.

## Как приобрести проверочные версии Windows

Компания Microsoft предоставляет две сборки Windows: проверочную (checked build) и свободную (free build). Рекомендуется выполнять тестирование и отладку драйверов на обеих версиях.

- ◆ **Проверочные версии.** Проверочные сборки Windows содержат отладочную информацию и позволяют применять определенные виды отладочных кодов, такие как, например, макросы ASSERT.

Проверочные сборки Windows подобны отладочным версиям пакетов для разработки приложений. Обычно проверочные сборки работают более медленно, чем окончательные версии. Так как некоторые опции оптимизации компилятора отключены, дизассемблированные машинные инструкции и трассировочные сообщения поддаются более легкой интерпретации.

- ◆ **Свободные сборки.** В свободных сборках Windows отсутствует отладочная информация, и они полностью оптимизированы.

Свободные сборки Windows подобны окончательным версиям пакетов для разработки приложений. Все потребительские версии Windows являются свободными сборками, т. к. этот тип сборки имеет самую лучшую производительность и занимает наименьший объем памяти.

Тестирования и отладку драйверов рекомендуется выполнять с помощью проверочных сборок Windows. Обычно разработчики переходят к проверке разрабатываемых драйверов на свободной сборке Windows в конце цикла разработки, после того как большинство ошибок были выявлены и устранены.

Проверочные сборки Windows можно получить с CD MSDN Subscriber или скачать из раздела **Subscription** страницы MSDN. Подробную информацию относительно этого вопроса можно найти в разделе **Using Checked Builds of Windows** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79304>.

## Как приобрести отладочные инструменты

Пакет Debugging Tools for Windows можно скачать бесплатно на Web-странице WHDC; он также входит в набор WDK.

### Чтобы приобрести пакет Debugging Tools for Windows на сайте WHDC:

1. Откройте страницу Debugging Tools for Windows на Web-сайте WHDC (<http://go.microsoft.com/fwlink/?LinkId=80065>).
2. Перейдите на страницу **Install** для 32-битной версии и следуйте инструкциям для скачивания пакета.
3. Перейдите на страницу **Install** для 64-битной версии и следуйте инструкциям для скачивания 64-битного пакета.

Информацию о том, куда устанавливать пакет, можно найти в его справочной документации.

Обычно, если отладчик исполняется на 32-битной системе, необходимо использовать 32-битную версию пакета. Если же отладчик исполняется на x64-системе, а операционная система на целевом компьютере — Windows XP или более поздняя версия Windows, можно использовать как 32-битную, так и 64-битную версию пакета. Более подробную информацию по этому вопросу можно найти в разделе "Choosing Between the 32-bit and 64-bit Packages"<sup>1</sup> в документации для пакета Debugging Tools for Windows.

---

<sup>1</sup> Какую версию пакета устанавливать, 32- или 64-битную. — *Пер.*

Пакет Debugging Tools for Windows содержит следующие компоненты.

- ◆ **Отладчики.** Для отладки как UMDF-, так и KMDF-драйверов мы рекомендуем пользоваться графическим отладочным инструментом WinDbg. Но разработчики, которые предпочитают работать с инструментами командной строки, могут также воспользоваться утилитой KD. Это консольное приложение обладает такими же возможностями, как и утилита WinDbg.
- ◆ **Коллекция связанных отладочных инструментов.** Пакет Debugging Tools for Windows также содержит другие отладочные инструменты. Например, утилита командной строки Tlist выводит на экран информацию об исполняемых процессах и может быть полезной при отладке драйверов UMDF. А с помощью утилиты DBN можно просмотреть идентификаторы в процессе отладки.
- ◆ **Отладочная документация.** Файл справки для Debugging Tools for Windows содержит инструкции по использованию отладочных инструментов и полную справку для стандартных команд и расширений отладчика.

При отладке драйверов, особенно драйверов режима ядра, обычно требуются файлы идентификаторов для версии Windows, под которой исполняется драйвер. Компания Microsoft предоставляет файлы идентификаторов для всех версий Windows. Для отладки драйверов режима ядра идентификаторы необходимо установить на компьютер с отладчиком, а не на компьютер с драйвером, подвергающимся отладке.

Идентификаторы для уровня аппаратных абстракций (FIAL) и для ядра Windows (KRNLL) устанавливаются совместно с установкой набора WDK в папку %wdk%\debug.

Чтобы получить самые последние идентификаторы, мы рекомендуем подключиться к серверу идентификаторов компании Microsoft, который загрузит идентификаторы автоматически.

**Чтобы получить идентификаторы Windows**, следуйте инструкциям в файле справки для отладчика WinDbg, чтобы подключиться к серверу символов компании Microsoft по адресу <http://msdl.microsoft.com/download/symbols>, с которого WinDbg автоматически скачает необходимые идентификаторы.

Либо скачайте текущие пакеты с Web-сайта WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79331>.

Использование идентификаторов при отладке драйверов, включая пользование сервером идентификаторов компании Microsoft (см. главу 22).

Ссылки на обучающие компании и другие ресурсы для обучения отладке драйверов Windows можно найти на Web-сайте WHDC в разделе **Debugging Tools and Symbols—Resources** (по адресу <http://go.microsoft.com/fwlink/?LinkId=79332>).

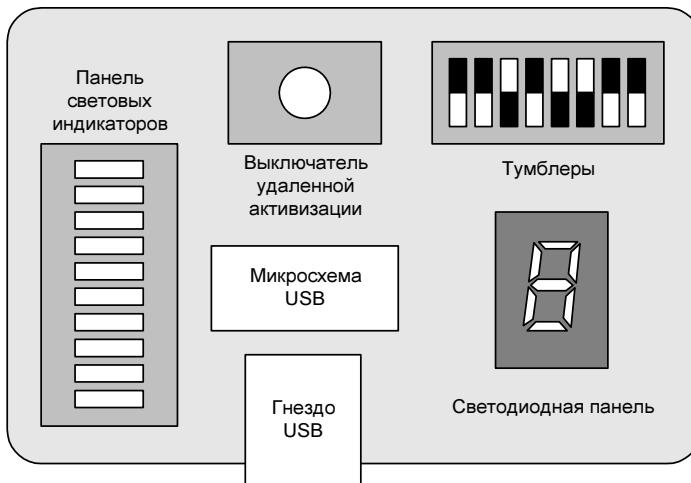
## Как приобрести устройства обучения OSR

Самый лучший способ обучения разработке драйверов устройств — это разработать драйвер для настоящего устройства. Но настоящие серийные устройства зачастую весьма сложные, что делает их не очень подходящими для начала обучения разработке драйверов. Кроме этого, технические спецификации, необходимые для создания драйверов для таких устройств, зачастую принадлежат компании — разработчику устройства. Вследствие этого, если вы не работаете на данного поставщика, получить их может быть трудной задачей.

С целью решения этой проблемы компания OSR разработала несколько обучающих наборов OSR (OSR learning kit), предназначенных специально для изучения разработки драйверов.

Эти устройства достаточно просты и позволяют начинающему разработчику фокусироваться на приобретении базовых навыков.

Физически, наборы представляют собой платы, которые или вставляются в разъем PCI, или подключаются к компьютеру с помощью кабеля USB. Платы визуально информируют об исполняющихся операциях, позволяя легко видеть, что происходит в данный момент. Например, они оборудованы светодиодным индикатором, который можно запрограммировать на вывод буквы и цифры (рис. 1.1). В этой книге используются образцы драйверов Microsoft, работающие с устройством OSR USB Fx2.



**Рис. 1.1.** Упрощенный рисунок обучающего устройства USB

Обучающие наборы OSR включают все необходимые для разработки драйвера технические спецификации на аппаратное оборудование плюс примеры кода и тестовые приложения. Дополнительную информацию об этих обучающих наборах и их приобретении можно найти на Web-сайте компании OSR по адресу <http://www.osronline.com>.

## Основные источники информации

Кроме образцов драйверов и документации набора WDK, многочисленные другие ресурсы можно найти на Web-сайте компании Microsoft и с помощью других сайтов и форумов в Интернете. Далее приводится несколько рекомендуемых источников таких ресурсов.

### ◆ WHDC (Windows Hardware Developer Central).

На Web-сайте WHDC собрана разнообразная коллекция ресурсов для разработчиков драйверов и изготовителей аппаратуры, включая доклады, учебные пособия, советы и образцы.

Здесь также имеются разные публикации, которые не входят в рамки WDK или MSDN, включая блоги экспертов из команды разработчиков WDF, в которых они обсуждают вопросы по разработке драйверов.

Подписавшись на Microsoft Hardware Newsletter (информационный бюллетень по аппаратному обеспечению от компании Microsoft), вы сможете получать извещения о новых докладах и комплектах, вместе с другой информацией, представляющей интерес для раз-

работчиков драйверов. Адрес Web-сайта WHDC в Интернете — <http://www.microsoft.com/whdc>.

#### ◆ Блоги.

Несколько членов команды разработчиков WDF ведут блоги, в которых они обсуждают различные аспекты разработки драйверов. Они часто отвечают на вопросы, обсуждаемые разработчиками драйверов на форумах и серверах рассылки. Список блогов экспертов Microsoft в области разработки драйверов можно просмотреть по адресу <http://go.microsoft.com/fwlink/?LinkId=79579>.

#### ◆ Группы новостей Microsoft для разработчиков драйверов.

С вопросами, на которые вы не можете найти ответы в документации WDK, можно обратиться к участникам одной из группы новостей MSDN. Очень часто эти группы новостей — единственный источник информации на многие трудноразрешимые проблемы. Вот список некоторых групп новостей по разработке драйверов:

- [Microsoft.public.development.device.drivers](mailto:Microsoft.public.development.device.drivers);
- [Microsoft.public.windowsxp.device\\_driver.dev](mailto:Microsoft.public.windowsxp.device_driver.dev);
- [Microsoft.public.win32.programmer.kernel](mailto:Microsoft.public.win32.programmer.kernel).

Доступ к этим группам можно получить с помощью браузера Internet Explorer или другого интернет-обозревателя. Можно также подключиться к серверу новостей с помощью Outlook Express или другого подобного приложения. Адрес сервера групп новостей — [msnews.microsoft.com](http://msnews.microsoft.com), а домашней страницы групп новостей Microsoft — <http://msdn.microsoft.com/newsgroups/>. Группы разработчиков драйверов находятся в разделе **Windows Development** Windows DDK.

В рамках бета-программ для WDF, компания Microsoft также содержит несколько групп новостей для KMDF, UMDF и инструментов.

Информацию о том, как можно принять участие в группах новостей WDF и о бета-программах WDF, можно найти в разделе **Hardware and Driver Developer Community** (сообщество разработчиков оборудования и драйверов) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79580>.

#### ◆ Компания OSR в Интернете.

Компания Open System Resources (OSR) содержит три важных сервера рассылок на своем Web-сайте:

- **NTDEV (Windows System Software Developers List)** — рассылка для разработчиков системного программного обеспечения Windows, посвященная разработке драйверов для операционных систем семейства Windows;
- **NTFSD (Windows File System Developers List)** — рассылка для разработчиков файловых систем Windows, посвященная разработке драйверов для файловых систем или драйверов-фильтров файловых систем;
- **WINDBG (Windows Debugger Users List)** — рассылка для пользователей отладчиками под Windows, посвященная вопросам, связанным с использованием и обновлениями отладчика WinDbg.

Информацию о том, как подписаться на эти рассылки, можно найти на Web-сайте компании OSR по адресу <http://www.osronline.com/>.

◆ **Channel 9.**

Web-сайт Channel 9, поддерживаемый MSDN, содержит ответы на обширный круг вопросов, включая видео, подкасты<sup>1</sup>, ресурсы формата wiki и форумы. Хотя большая часть этого сайта отведена вопросам разработки приложений, здесь также имеются некоторые материалы по разработке драйверов, внутренней организации Windows и аппаратным средствам. Адрес сайта — <http://channel9.msdn.com/>.

◆ **Конференции.**

Компания Microsoft проводит конференции, посвященные интересам разработчиков драйверов и производителей аппаратного обеспечения. Также несколько организаций и частных компаний, занимающиеся разработкой драйверов и связанными вопросами, предоставляют профессиональное обучение в этой области. Дополнительную информацию по этому вопросу можно найти в разделе **Hardware and Driver Developer Community** (сообщество разработчиков драйверов и производителей оборудования) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79580>.

◆ **Конференция WinHEC и другие мероприятия, организуемые компанией Microsoft.**

Конференция WinHEC является основным из мероприятий, организуемых компанией Microsoft для разработчиков аппаратного обеспечения, разработчиков и тестировщиков драйверов, а также для лиц, принимающих деловые решения, и специалистов по товарному планированию. Конференция обычно проводится весной каждого года.

Информацию о конференции см. на Web-сайте по адресу <http://www.microsoft.com/whdc/winhec/>.

◆ **Курсы и семинары.**

Компания Microsoft и несколько других компаний предоставляют различные курсы и семинары на темы, представляющие интерес для разработчиков драйверов.

Для информации по этому вопросу см. раздел **Conferences and Training for Developers** (конференции и профессиональное обучение для разработчиков) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79334>.

## Основные ссылки

- ◆ WDF на WHDC — <http://go.microsoft.com/fwlink/?LinkId=79335>;
- ◆ WDK на WHDC — <http://go.microsoft.com/fwlink/?LinkId=79337>;
- ◆ блоги разработчиков драйверов — <http://go.microsoft.com/fwlink/?LinkId=79579>;
- ◆ группы новостей — <http://go.microsoft.com/fwlink/?LinkId=79580>.

### Как быть в курсе обновлений WDF?

Мы постоянно размещаем новую информацию о WDF и ее инфраструктурах на Web-сайте WHDC. Регулярно проверяйте страницу **Developing Drivers with WDF: News and Updates** на Web-сайте WHDC на наличие новых образцов кода, докладов, новостей о драйверных инфраструктурах и наборе WDK и любых обновлений этой книги. Здесь же размещается новая информация о блогах разработчиков драйверов для Windows. Мы также разместили все ссылки на весь справочный материал в этой книге по адресу <http://go.microsoft.com/fwlink/?LinkId=80911> (см. **Hot Links for WDF References**), так что вы сможете быстро перейти к необходимому справочному материалу по этим ссылкам, не вводя их с клавиатуры. (*Интернет-команда WHDC, компания Microsoft.*)

---

<sup>1</sup> От англ. *podcast* — цифровая запись радио- или телепрограммы, которую можно скачать из Интернета. — Пер.

## ГЛАВА 2

# Основы драйверов под Windows

Эта глава предназначена для программистов, которые раньше не занимались разработкой драйверов под Windows. В ней предоставляется основная информация о ядре операционной системы Windows и о том, каким образом драйверы работают в этой среде. Также дается вводная информация о программировании в режиме ядра.

Если вы новичок в области разработки драйверов Windows, вам нужно прочитать эту главу, т. к. в ней представляются понятия и терминология, являющиеся необходимыми для понимания дальнейших тем, рассматриваемых в этой книге. Если же у вас имеется опыт разработки драйверов, вы можете ограничиться поверхностным просмотром этой главы, лишь уделив внимание разд. *"Основные термины"* в конце этой главы. Если же вы знакомы со всеми понятиями и терминами, изложенными в этой главе, то можете полностью пропустить ее.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Каталог образцов набора WDK	%wdk%\НомерСборки\src
<b>Документация WDK</b>	
Getting Started with Windows Drivers <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79284">http://go.microsoft.com/fwlink/?LinkId = 79284</a>
<b>Прочее</b>	
Driver Fundamentals Resources <sup>2</sup> на сайте HDC	<a href="http://go.microsoft.com/fwlink/?LinkId=79338">http://go.microsoft.com/fwlink/?LinkId = 79338</a>
Microsoft Windows Internals (авторы — Solomon и Russinovich) <sup>3</sup>	<a href="http://www.microsoft.com/MSPress/books/6710.aspx">http://www.microsoft.com/MSPress/books/6710.aspx</a>
Operating Systems (Stallings)	<a href="http://go.microsoft.com/fwlink/?LinkId=82718">http://go.microsoft.com/fwlink/?LinkId = 82718</a>

<sup>1</sup> Приступая к изучению драйверов под Windows. — Пер.

<sup>2</sup> Ресурсы по основам драйверов. — Пер.

<sup>3</sup> Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. / Пер. с англ. — 4-е изд. — Издательско-торговый дом "Русская Редакция"; СПб.: Питер; 2005. — 992 с.

## Что такое драйвер?

Ядро Windows не предназначено для самостоятельного взаимодействия с устройствами. Для обнаружения подсоединенных устройств, обмена информацией с устройствами и предоставления возможностей драйвера клиентам, таким как приложения, ядро Windows полагается на драйверы устройств. Windows же предоставляет абстрактный интерфейс для поддержки устройств, называемый *драйверная модель*. Задача разработчиков драйверов состоит в реализации данного интерфейса, чтобы поддерживать конкретные требования аппаратного устройства.

Иными, более простыми, словами, обычно задачей драйвера является предоставление связи и контролирование взаимодействия между приложениями и устройством. Во многих отношениях работа драйверов сходна с работой сервисов. Так, например, драйверы:

- ◆ исполняются в фоновом режиме, отдельно от процессов приложений, и доступны множественным пользователям;
- ◆ имеют долгое время жизни — такое же, как и время жизни соответствующих им устройств. Драйвер устройства запускается, когда Windows обнаруживает устройство, и прекращает исполнение, когда устройство удалено;
- ◆ отвечают на внешние запросы ввода/вывода. Эти запросы обычно исходят от приложений, Windows и иногда от других драйверов;
- ◆ не имеют пользовательского интерфейса. Пользователи обычно взаимодействуют с драйвером, указывая приложению издать запрос ввода/вывода;
- ◆ исполняются в другом адресном пространстве, чем адресное пространство приложения, издавшего запрос ввода/вывода.

Но драйверы отличаются от сервисов во многих важных аспектах. Так, они:

- ◆ взаимодействуют с основными сервисами Windows и устройствами посредством своих собственных специальных интерфейсов, которые называются DLL;
- ◆ основаны на модели ввода/вывода Windows, совершенно иной, чем модель, применяемая для сервисов и приложений;
- ◆ могут напрямую взаимодействовать с компонентами режима ядра. Драйверы режима ядра исполняются полностью в режиме ядра. Но даже драйверы UMDF иногда должны взаимодействовать с базовыми драйверами режима ядра для обмена данными с устройством.

В этой главе предоставляется концептуальное описание расположения драйверов в структуре операционной системы Windows и то, каким образом они управляют потоком запросов между устройствами и их клиентами. Хотя эта книга посвящена драйверам типа WDF, данная глава в основном сфокусирована на более старой модели WDM, которая основана на интерфейсе драйвера устройства (в дальнейшем *интерфейс DDI* — driver device interface), предоставляемого непосредственно ядром Windows. Модель WDM предусматривает большую гибкость, но для разработчиков программного обеспечения разработка драйверов с применением этой модели оказалась трудной задачей. Тем не менее важно понимать, по крайней мере, основы модели WDM.

- ◆ Модель WDF разработана, чтобы заменить модель WDM в качестве главной драйверной модели Windows, предоставляя уровень абстракции над моделью WDM; механизм WDM продолжает работать в фоне. Чтобы понять модель WDF, необходимо понимать некоторые базовые концепты модели WDM.

- ♦ На концептуальном уровне драйверы обеих инфраструктур, WDF и WDM, имеют подобную структуру и обрабатывают запросы ввода/вывода одинаковым образом. Большая часть обсуждения в этой главе применима как к драйверам WDM, так и к драйверам WDF, хотя детали реализации отличаются для каждой модели.

Эта глава сосредоточена на драйверах и методах программирования режима ядра, т. к. все разработчики драйверов должны иметь базовое понимание концепции режима ядра. Но понимание основ режима ядра будет также полезным и для разработчиков, заинтересованных в драйверах пользовательского режима. Например, структура драйвера UMDF сходна со структурой драйверов WDM и KMDF, а драйверы UMDF обрабатывают запросы ввода/вывода в большей части таким же образом, как и драйверы режима ядра.

## Базовая архитектура Windows

Лучшей отправной точкой для изучения работы драйверов будет обсуждение, какое место драйверы занимают в базовой архитектуре операционной системы Windows. Архитектура Windows состоит из нескольких уровней, при этом приложения находятся на верхних уровнях, а аппаратная часть — на нижних. На рис. 2.1 приведена упрощенная схема архитектуры Windows, на которой показано, каким образом драйверы включены в общую архитектуру.

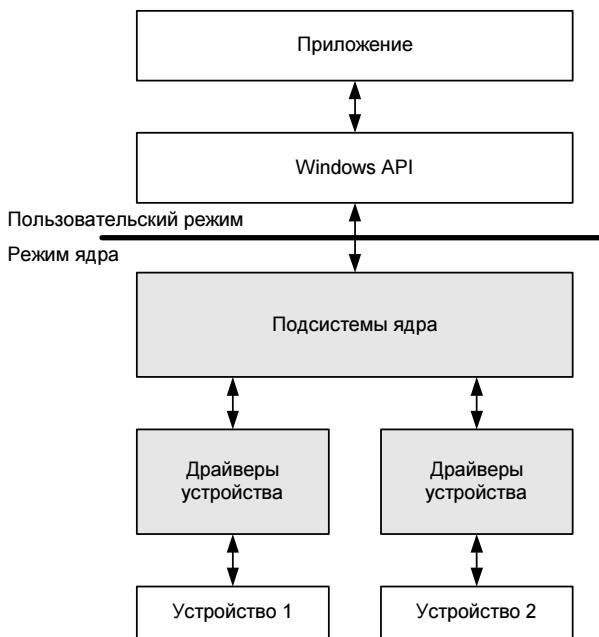


Рис. 2.1. Основная архитектура операционной системы Windows

Рассмотрим основные компоненты архитектуры.

**Приложения и Windows API.** Приложения обычно инициируют запросы ввода/вывода. Но приложения и сервисы исполняются в пользовательском режиме и не имеют прямого доступа к компонентам режима ядра. Поэтому они дают запросы ввода/вывода путем вызова соответствующих функций Windows API, которые, в свою очередь, передают эти запросы соответствующим компонентам ядра с помощью таких составляющих Windows, как NtDll.dll.

**Пользовательский режим и режим ядра.** Windows работает в двух отчетливо различающихся режимах — пользовательском режиме и режиме ядра. Приложения исполняются в пользовательском режиме, а центральная часть операционной системы, называемая *ядром* и включающая все драйверы режима ядра, исполняется в режиме ядра. Между возможностями и связанными рисками этих двух режимов существует четкая разница.

- ◆ **Приложения пользовательского режима не имеют доверительных отношений с ядром Windows.** Они исполняются в ограниченной среде, которая не позволяет им нарушить целостность других приложений или ядра операционной системы.
- ◆ **Приложения режима ядра, включая драйверы режима ядра, являются доверяемыми компонентами ядра Windows.** На их исполнение накладываются лишь относительно небольшие ограничения, которые сопровождаются некоторыми соответствующими рисками.

Граница между пользовательским режимом и режимом ядра в некоторой степени работает подобно одностороннему зеркалу<sup>1</sup>. Приложения могут взаимодействовать с ядром только косвенным образом, посредством интерфейса Windows API. Когда приложение выдает запрос ввода/вывода, оно не может "заглянуть" в ядро, чтобы непосредственно передать запрос драйверу режима ядра. С другой стороны, драйверы режима ядра могут "смотреть" в пользовательский режим и передавать данные непосредственно приложениям. Но этот подход представляет угрозу безопасности и имеет лишь ограниченное применение. Обычно драйвер режима ядра возвращает данные приложению косвенно, посредством одной из подсистем ядра.

**Подсистемы ядра.** Подсистемы ядра реализуют большую часть основной функциональности Windows. Они предоставляют подпрограммы интерфейса DDI, которые позволяют драйверам взаимодействовать с системой. Наиболее часто драйверы взаимодействуют со следующими подсистемами.

- ◆ **Менеджер ввода/вывода (I/O Manager).** Облегчает взаимодействие между приложениями и устройствами. Менеджер ввода/вывода получает запросы от пользовательского режима, после чего передает их соответствующему драйверу. Он также получает завершенные запросы ввода/вывода от драйвера и передает любые данные назад пользовательскому режиму и, в конечном счете, приложению, которое выдало запрос.
- ◆ **PnP-менеджер.** Обрабатывает задачи Plug and Play, такие как обнаружение устройств, подключаемых к шине, создание стека для каждого устройства, и обрабатывает процесс добавления и удаления устройств при работающей системе.
- ◆ **Менеджер энергопотребления.** Обрабатывает изменения в энергосостоянии компьютера, например, переход из рабочего состояния при полном энергопотреблении в состояние сна или наоборот.

Другие подсистемы ядра, предоставляющие подпрограммы DDI, включают менеджер памяти и менеджер потоков и процессов. Хотя подсистемы ядра являются самостоятельными компонентами, между ними существуют сложные зависимости. Например, поведение PnP-менеджера зависит от состояния энергопотребления и наоборот.

**Драйверы и устройства.** Драйверы предоставляют интерфейс между своими устройствами и подсистемами ядра. Подсистемы ядра выдают запросы ввода/вывода драйверам, которые

---

<sup>1</sup> Зеркало, прозрачное с одной стороны, используемое, в особенности, для скрытого наблюдения. — Пер.

обрабатывают эти запросы и взаимодействуют с соответствующими устройствами необходимым образом. Любые данные, которые драйвер получает от устройства, обычно передаются обратно подсистеме ядра, которая генерировала данный запрос. Драйверы также отвечают на запросы, издаваемые подсистемами ядра, такими как PnP-менеджер или менеджер энергопотребления, для обработки таких задач, как подготовка устройства для удаления или перехода в режим сна.

Для подробного рассмотрения архитектуры Windows см. книгу "Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000"<sup>1</sup>. Информацию о том, как приобрести книгу, см. по адресу <http://www.microsoft.com/MSPress/books/6710.aspx>.

## Архитектура драйвера

Архитектура драйверов Windows спроектирована для поддержки четырех ключевых возможностей.

- ◆ **Модульная многоуровневая архитектура.** Устройство могут обслуживать несколько драйверов, организованных в стек.
- ◆ **Пакетный механизм ввода/вывода.** Все запросы обрабатываются в пакетах, которыми передаются между системой и драйвером, и от одного драйвера в стеке другому.
- ◆ **Асинхронный ввод/вывод.** Драйверы могут начать обрабатывать другие запросы, не ожидая завершения обработки текущего запроса.
- ◆ **Динамическая загрузка и выгрузка.** Время жизни драйвера привязано к времени жизни его устройства. Драйвер выгружается только после того, как все связанные устройства были удалены.

## Объекты устройств и стек устройства

Запрос ввода/вывода, направленный подсистемой ядра устройству, обрабатывается одним или несколькими драйверами. Каждый драйвер имеет ассоциированный с ним объект устройства, который представляет участие драйвера в обработке запросов ввода/вывода данного устройства. Объект устройства — это структура данных, содержащая указатели на рабочие процедуры, которые позволяют менеджеру ввода/вывода взаимодействовать с драйвером.

Объекты устройств организованы в стек устройства, при этом для каждого устройства создается отдельный стек. Обычно под термином "*стек устройства*" подразумевается стек объектов устройства с их связанными драйверами. Но стек устройства ассоциируется только с одним устройством, в то время как набор драйверов может обслуживать несколько стеков устройств. Набор драйверов иногда называется *стеком драйверов*.

На рис. 2.2 показаны два устройства (каждый со своим стеком устройства), которые обслуживаются одним набором драйверов.

---

<sup>1</sup> Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. / Пер. с англ. — 4-е изд. — Издательско-торговый дом "Русская Редакция"; СПб.: Питер; 2005. — 992 с.

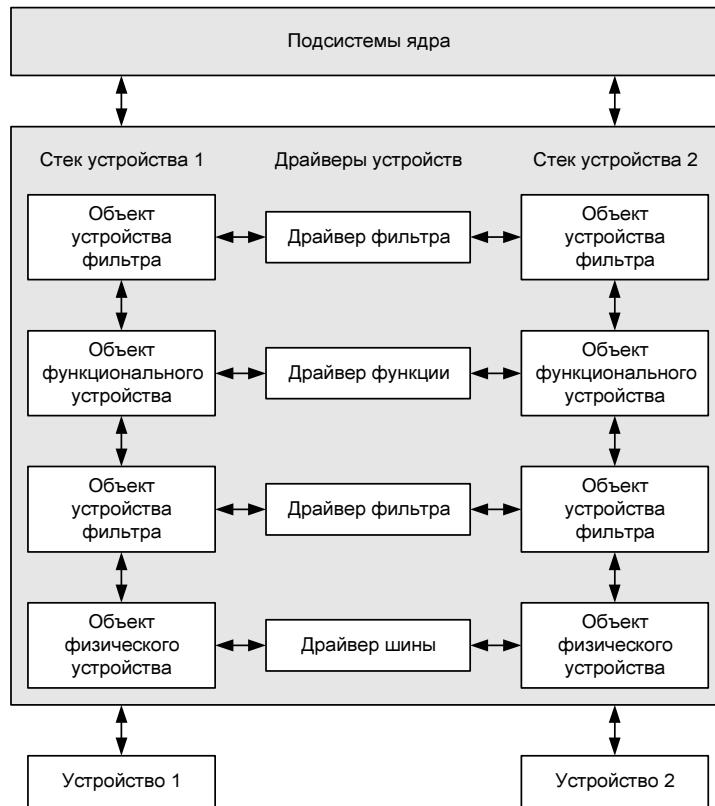


Рис. 2.2. Стек устройства

Стек устройства состоит из следующих компонентов.

- ◆ **Драйвер шины и объект физического устройства.** Внизу стека находится объект физического устройства (в дальнейшем PDO<sup>1</sup>), который ассоциирован с драйвером шины. Устройства обычно подключаются к стандартным аппаратным шинам, таким как, например, PCI или USB. Драйвер шины обычно управляет несколькими аппаратными устройствами, подключенными к физическойшине.

Например, после установки драйвер шины перечисляет устройства, подключенные кшине, и запрашивает ресурсы для этих устройств. PnP-менеджер использует эту информацию для выделения ресурсов каждому устройству. Каждое устройство имеет собственный объект PDO. PnP-менеджер идентифицирует драйверы для каждого устройства и создает соответствующий стек устройств вверху каждого объекта PDO.

- ◆ **Драйвер функции и объект функционального устройства.** Основным компонентом стека устройств является *объект функционального устройства* (в дальнейшем *объект FDO* (Functional Device Object)), который ассоциируется с драйвером функции. Драйвер функции преобразует абстракцию устройства, создаваемую Windows, в настоящие команды, необходимые для обмена данными с физическим устройством. Он предостав-

<sup>1</sup> PDO (Physical Device Object) — объект физического устройства. В некоторых публикациях переводится как *физический объект устройства*. Абсурдность такого толкования очевидна. — Пер.

ляет так называемую "верхнюю кромку" (иными словами, интерфейс устройства) для взаимодействия с приложениями и устройствами, и обычно определяет, каким образом драйвер реагирует на изменения в состоянии Plug and Play или энергопотребления. Так называемая "нижняя кромка" драйвера устройства обрабатывает взаимодействие с устройством или другими драйверами, такими как, например, расположенный ниже драйвер фильтра или драйвер шины.

- ◆ **Драйверы фильтров и объекты устройства фильтра.** Стек устройства может иметь несколько объектов устройства фильтра (в дальнейшем *объект FiDO* (Filter Device Object)), которые могут располагаться вокруг объекта FDO. С каждым объектом FiDO ассоциируется драйвер фильтра. Драйверы фильтров не являются обязательными, но часто присутствуют. Сторонние поставщики могут снабжать стек устройства драйверами фильтров с целью придания ему дополнительных возможностей, таким образом, повышая его стоимость. Обычно драйвер фильтра служит для модификации некоторых запросов ввода/вывода, когда они проходят через стек устройств, в значительной степени подобно тому, как аудиофильтры модифицируют аудиопоток.

Например, с помощью драйверов фильтра можно зашифровывать или расшифровывать запросы на чтение и запись. Драйверы фильтра можно также применять для выполнения задач, не требующих модификации запросов ввода/вывода, например, таких как отслеживание запросов и предоставление информации обратно отслеживающему приложению.

Эти три типа объектов устройств отличаются в деталях, но работают по обеспечению системы возможностями обработки запросов ввода/вывода в основном одинаково. Каким образом стек устройств обрабатывает запросы ввода/вывода, рассматривается в разд. "Объекты и структуры данных ядра" и "Модель ввода/вывода Windows" далее в этой главе.

## Дерево устройств

Шины компьютерных систем обычно организованы в несколько уровней. К шине самого низкого уровня может быть подключена одна или несколько других шин, а также индивидуальные устройства. К этим шинам, в свою очередь, тоже могут подключаться другие шины и независимые устройства, и т. д. Поэтому большинство драйверов шин должны играть две роли: драйвера функции для низележащей шины и перечислителя и менеджера шины для любых подключенных устройств.

При загрузке системы PnP-менеджер начинает работу с шинами самого низкого уровня и выполняет следующие операции:

1. Загружает драйвер шины, который перечисляет объекты PDO для всех устройств, подключенных к его физическойшине, и запрашивает ресурсы для этих устройств.
2. Выделяет ресурсы каждому устройству.
3. Опрашивает свою базу данных, чтобы определить ассоциирование драйверов с устройствами.
4. Создает стек устройств поверх каждого объекта PDO.
5. Запускает все устройства.
6. Опрашивает каждое устройство на возможное наличие у него объектов PDO для перечисления.
7. При необходимости повторяет шаги 2—6.

Результаты этого процесса представляются иерархической структурой, которая называется *деревом устройств*. Каждый стек устройства представляется в дереве устройств узлом, который называется *devnode*. Узел devnode устройства используется PnP-менеджером для хранения информации о конфигурации и отслеживания устройства. Основа дерева устройств — это абстрактное устройство, называемое *корневым устройством*.

Пример дерева устройств показан на рис. 2.3.

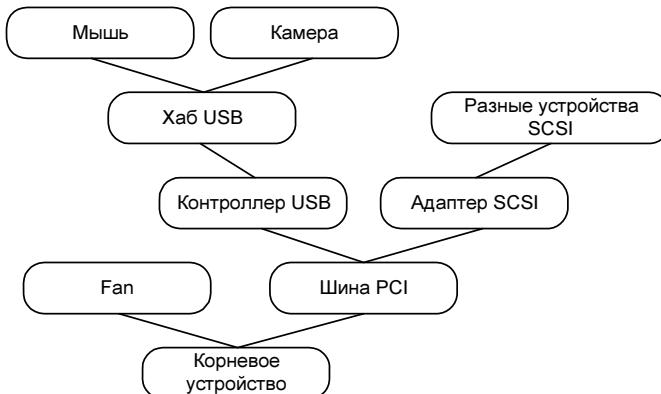


Рис. 2.3. Пример дерева устройств Plug and Play

## Объекты и структуры данных ядра

Для выполнения своих задач драйверы зависят от различных объектов и структур данных ядра. Эти объекты и структуры данных управляются ядром Windows и инкапсулируют различные системные ресурсы, такие как устройства, запросы ввода/вывода, потоки, события и т. д.

- ◆ **Объекты.** Объекты управляются менеджером объектов и отслеживаются методом подсчета всех ссылок на них (reference count). Например, объект `\KernelObjects\LowMemoryCondition` — это стандартный событийный объект, который сигнализирует нехватку памяти. Менеджер объектов также управляет процессами и потоками. Многие объекты могут иметь дескрипторы, с помощью которых приложения могут манипулировать объектом.
- ◆ **Структуры данных.** Большая часть используемых драйверами данных — это структуры данных, управляемые ядром. Хотя они и не являются объектами в том же смысле, как объекты типа `\KernelObjects\LowMemoryCondition`, эти структуры данных часто называются объектами и представляются подобно объектам. Например, для манипулирования, по крайней мере, некоторыми аспектами структур данных ядра драйверам часто необходимо использовать методы. Типичным примером структуры данных ядра может быть пакет запроса ввода/вывода (в дальнейшем *пакет IRP* (I/O Request Packet)), с помощью которого стеку устройства передаются запросы ввода/вывода и подобные запросы.

Для получения информации об именованных объектах см. раздел **Object Management** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82272>.

## Модель ввода/вывода Windows

Модель ввода/вывода Windows определяет, каким образом система и связанные драйверы обрабатывают запросы ввода/вывода. Модель охватывает не только обмен данными с устройством, она также работает как общий пакетный механизм взаимодействия между клиентами, выдающими запросы к стеку устройств. Клиентами являются приложения, подсистемы ядра, такие как PnP-менеджер, и сами драйверы.

Все запросы ввода/вывода Windows исполняются посредством пакетов IRP. Кроме исполнения запросов, связанных с вводом/выводом, которые приложения посыпают драйверам (например, запросы на чтение, запись, создание и т. д.), пакеты IRP также применяются для исполнения запросов Plug and Play, управления энергопотреблением и интерфейса WMI (Windows Management Instrumentation, инструментарий управления Windows), а также для передачи извещений об изменениях состояния устройств и обращений за информацией о ресурсах устройств и драйверов.

Обычно приложения обращаются к устройству синхронным образом. Например, когда приложение делает запрос к драйверу на чтение данных из устройства, оно ожидает возврата данных, после чего использует полученные данные каким-либо образом. Потом приложение запрашивает новые данные, снова использует их в своей работе, и так до тех пока, пока все данные не будут считаны.

Но с точки зрения драйверов, процесс ввода/вывода Windows является по существу асинхронным. Например, получив от приложения запрос на чтение, если драйвер может возвратить запрошенные данные в разумный (с его точки зрения) промежуток времени, то он так и делает. Но в некоторых случаях для исполнения запроса требуется значительное (с точки зрения драйвера) время. В таком случае драйвер начинает необходимую операцию и сообщает менеджеру ввода/вывода, что запрос находится в процессе исполнения. Тем временем драйвер может исполнять вновь поступающие запросы или возвращаться к обработке ранее запущенных на исполнение запросов.

Приложения часто применяют синхронный ввод/вывод. В таком случае менеджер ввода/вывода просто ожидает, пока драйвер не закончит операцию чтения, чтобы возвратить запрошенные данные приложению. Но приложение может воспользоваться тем обстоятельством, что драйвер обрабатывает ввод/вывод асинхронно, и также выполнять операции ввода/вывода асинхронно. Получив извещение от драйвера об ожидаемом завершении запроса, менеджер ввода/вывода извещает об этом приложение и возвращает ему управление потоком. Пока запрос выполняется, приложение может использовать этот поток для выполнения других задач.

По завершению выполнения операции чтения устройство информирует драйвер об этом. Драйвер сообщает об этом менеджеру ввода/вывода, который, в свою очередь, информирует приложение, и приложение потом может обрабатывать данные. Асинхронный режим ввода/вывода может существенно повысить производительность многопоточных приложений, т. к. приложение не тратит понапрасну время или потоки, ожидая завершения отнимающих много времени запросов ввода/вывода.

### Асинхронный ввод/вывод?

Порой возникает вопрос, каким образом драйвер может знать, какой тип ввода/вывода приложение запрашивает, синхронный или асинхронный? Одним из достоинств модели ввода/вывода Windows является то, что вам не требуется знать этого, т. к. в обоих случаях можно применить одинаковую процедуру. Если данные имеются в немедленном наличии,

то запрос удовлетворяется на месте. В противном случае, запускается процесс извлечения данных (запуск на исполнение команды, запись данных и т. д.) и управление возвращается менеджеру ввода/вывода. Во многих отношениях, это упрощает драйвер устройств. (*Петер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Запросы ввода/вывода

Модель ввода/вывода Windows отражает многослойную архитектуру, показанную на рис. 2.1. Клиентами, наиболее часто пользующимися сервисами драйверов, являются приложения пользовательского режима. Они издают запросы ввода/вывода посредством получения дескриптора устройства и, вызывая соответствующую функцию Windows. Запрос передается вниз по системе, пока он не дойдет до драйвера и, в конце концов, до устройства. Система потом возвращает полученный ответ приложению.

Наиболее распространенными типами запросов ввода/вывода являются следующие.

- ◆ **Запросы на запись.** Эти запросы передают драйверу данные для записи в устройство.
- ◆ **Запросы на чтение.** Эти запросы передают драйверу буфер для заполнения данными из устройства.
- ◆ **Запросы управления вводом/выводом устройства (IOCTL).** Эти запросы применяются для взаимодействия с драйверами, иного, нежели чтение или запись данных. С некоторыми устройствами запросы управления вводом/выводом являются наиболее применяемым типом запросов.

Приложения выдают эти запросы ввода/вывода с помощью стандартных функций Windows WriteFile, ReadFile И DeviceIoControl.

Для получения информации о том, как приложение получает дескриптор устройства, см. раздел **Device Management** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82273>.

## Обработка прерываний стеком устройства

Когда клиент посыпает устройству запрос ввода/вывода, менеджер ввода/вывода упаковывает запрос и связанную информацию, например буфера данных, в пакет IRP. Менеджер ввода/вывода находит указатель на соответствующую рабочую процедуру в верхнем объекте устройства в стеке и вызывает данную процедуру, после чего передает пакет IRP соответствующему драйверу для обработки. На рис. 2.4 показана схема обработки пакетов IRP стеком устройства.

Если верхний драйвер может удовлетворить данный IRP, он дает инструкцию менеджеру ввода/вывода выполнить его. Это действие возвращает пакет IRP менеджеру ввода/вывода, который, в свою очередь, передает запрошенные данные обратно клиенту. Но драйвер не всегда может удовлетворить запрос собственными силами. В таком случае драйвер обрабатывает пакет IRP должным образом и потом указывает менеджеру ввода/вывода передать пакет IRP следующему драйверу ниже в стеке устройства. Этот драйвер выполняет те операции, которые может, для удовлетворения запроса, и передает запрос ниже по стеку, и т. д. Например, драйвер фильтра может модифицировать пакет IRP на запись, после чего передать его ниже по стеку, чтобы драйвер функции мог передать данные устройству и завершить запрос. Иногда пакеты IRP приходится передавать до самого драйвера шины.

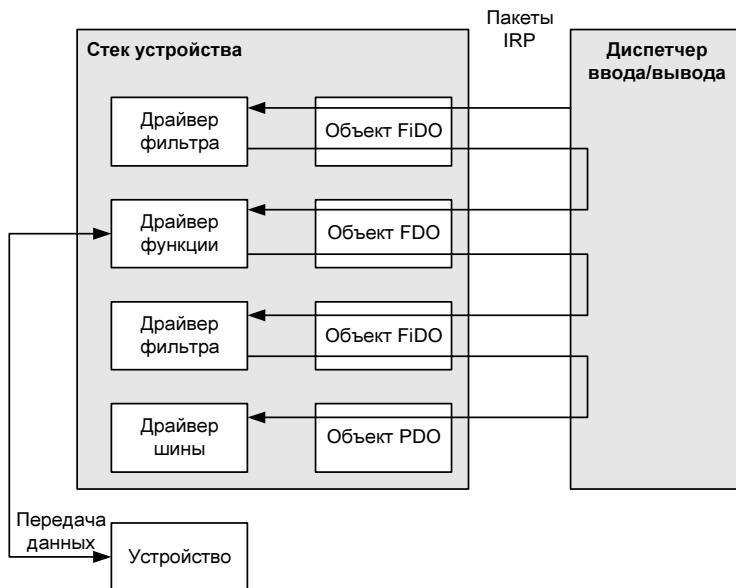


Рис. 2.4. Обработка пакетов IRP стеком устройства

В конце концов, запрос достигает того драйвера, который может удовлетворить и завершить его. Для удовлетворения запроса драйверам часто необходимо взаимодействовать с устройством или напрямую через аппаратные регистры или косвенно посредством драйвера протокола нижнего уровня. Иногда это взаимодействие осуществляется быстро, и драйвер может немедленно возвратить клиенту запрошенные данные. Но чаще при взаимодействии с устройством проходит много времени (исчисляемого в циклах центрального процессора), прежде чем устройство ответит на запрос. В таких случаях драйвер может возвратить управление клиенту, оставив запрос в процессе ожидания завершения исполнения, и завершает запрос по окончании процесса взаимодействия.

При передаче драйвером запроса вниз по стеку может оказаться необходимым выполнить какую-либо дополнительную обработку после завершения запроса, но перед тем, как возвращать результаты клиенту. В таком случае перед тем, как вызывать менеджер ввода/вывода для передачи пакета IRP следующему драйверу, драйвер может установить процедуру завершения ввода/вывода. Когда запрос, наконец, завершен, менеджер ввода/вывода вызывает процедуры завершения в порядке, обратном их установке, т. е. "разматывается" обратно вверх по стеку.

Процесс ввода/вывода обсуждается в главе 8.

## Буферы данных и типы передачи ввода/вывода

В пакетах IRP передаются три типа информации:

- ◆ управляющая информация, например, тип запроса;
- ◆ информация о статусе обработки запроса;
- ◆ буферы данных для передачи драйвером от клиента устройству и в обратном направлении. В частности, в этих буферах содержатся данные, которые необходимо передать от клиента устройству или от устройства клиенту.

Обычно тип данных в буферах безразличен драйверу, он только должен передать байты устройству или от него. Драйверы получают доступ к буферам данных, связанных с пакетом IRP, одним из следующих способом.

- ◆ **Буферизованный ввод/вывод (buffered I/O).** Менеджер ввода/вывода создает системный буфер такого же размера, как и буфер, созданный клиентом, и копирует данные клиента в этот буфер. После этого менеджер ввода/вывода передает драйверу в поле `AssociatedIrp.SystemBuffer` пакета IRP указатель на системный буфер, и драйвер выполняет операции чтения и записи с системным буфером. Этот метод передачи иногда называется `METHOD_BUFFERED`.
- ◆ **Прямой ввод/вывод (direct I/O).** Windows блокирует в памяти буфер клиента. После этого драйверу передается список MDL (Memory Descriptor List, список дескрипторов памяти), в котором перечислены страницы памяти, составляющие буфер. С помощью списка MDL драйвер осуществляет доступ к страницам. Этот метод передачи иногда называется `METHOD_DIRECT`.
- ◆ **Ни буферизованный, ни прямой ввод/вывод.** Windows передает размер буфера и его начальный адрес в адресном пространстве клиента. Этот метод передачи иногда называется `METHOD_NEITHER`, или, более просто, *neither I/O*.

Клиент и драйвер исполняются в разных адресных пространствах, поэтому драйвер должен осуществлять доступ к буферу с осторожностью. В частности, драйвер не может просто разыменовать указатель на буфер пользовательского режима с какой-либо уверенностью в том, что данные, находящиеся по этому адресу, являются действительными, или даже в том, что сам указатель является действительным.

Перечислим уровни риска этих трех типов передачи.

- ◆ Буферизованный ввод/вывод — безопасный по своей природе.  
Система гарантирует действительность буфера, создавая его в адресном пространстве режима ядра, копируя и вставляя данные в пользовательский режим.
- ◆ Прямой ввод/вывод представляет некоторую опасность.  
Система гарантирует действительность адресов пользовательского режима в буфере, блокируя страницы в памяти. Но при этом режиме нет гарантии, что данные по этим адресам не будут модифицированы приложением.
- ◆ Режим ввода/вывода NEITHER по своей природе ненадежный.

В данном режиме драйверу просто передается адрес пользовательского режима буфера. У драйвера даже нет гарантии, что со временем, когда он попытается осуществить доступ к буферу, этот указатель будет верным. Драйвер может осуществлять безопасный доступ к буферу пользовательского режима, только если он предпримет меры для обеспечения действительности указателя на буфер.

Каждый из этих способов ввода/вывода обсуждается в главе 8.

## Передача и получение данных от устройства

Часто запросам ввода/вывода необходимо передать или получить данные от устройства. Обычно передача данных обрабатывается драйвером функции устройства. О завершении передачи устройство обычно извещает драйвер, генерируя прерывание. На наиболее общем уровне прерывание сообщает системе о событии, которое требует немедленной обработки.

Само название "прерывание" происходит из обстоятельства, что система должна прервать нормальную обработку потоков, чтобы немедленно обработать событие.

Детали процесса передачи данных зависят от типа устройства.

- ◆ В случае устройств, работающих на основе протоколов, например USB-устройства, и некоторых других типов устройств, процесс передачи данных устройству обрабатывается системными драйверами.

Драйвер функции упаковывает буфер в необходимом формате и передает его системному драйверу. Этот драйвер управляет передачей данных и обрабатывает прерывание. После этого системный драйвер возвращает управление вместе с полученными данными драйверу функции.

- ◆ В случае устройств иных типов драйвер должен самостоятельно обрабатывать процесс передачи данных устройству или от него, используя для этого механизм DMA.

Драйвер также должен реализовать процедуру ISR (Interrupt Service Routine, процедура обслуживания прерывания) для обработки прерывания, которое генерируется по завершению транзакции.

Прерывания рассматриваются в *главе 16*, а в *главе 17* обсуждаются методики DMA.

## Plug and Play и управление энергопотреблением

Но не все запросы связаны с операциями ввода/вывода. Запросы Plug and Play и управления энергопотреблением извещают драйвер о разнообразных событиях, связанных с выделением ресурсов, нахождением устройств, установкой и загрузкой устройств, загрузкой драйверов и изменениями в состоянии энергопотребления. Вот несколько примеров типичных событий:

- ◆ компьютер выходит из состояния сна;
- ◆ пользователь физически подключает новое устройство по горячему;
- ◆ пользователь физически удаляет подключенное устройство по горячему;
- ◆ компьютер уменьшает энергопотребление и переходит в состояние сна.

На практике Plug and Play и события энергопотребления тесно связаны и требуют совместного управления. Запросы Plug and Play и управления энергопотреблением не являются вводом/выводом в строгом смысле этого термина, но эти запросы упаковываются в форме пакетов IRP и проходят через драйвер в большей степени таким же образом, как и запросы на чтение, запись или управления вводом/выводом устройства. Управляет механизмом Plug and Play и состоянием энергопотребления и удовлетворяет соответствующие пакеты IRP обычно драйвер функции. Драйверы фильтров чаще всего только передают такие запросы вниз по стеку.

Обработка этих типов запросов механизмом WDF рассматривается в *главе 7*.

## Основы программирования в режиме ядра

Программирование в режиме ядра является предметом, без знания которого разработчику драйверов не обойтись. Но некоторые аспекты программирования в режиме ядра существенно отличаются от программирования в пользовательском режиме. Кроме этого, при программировании в режиме ядра часто приходится выполнять привычные операции другим способом, и при этом довольно часто требуется быть намного осторожнее.

Например, когда объем используемой памяти превышает существующую физическую память, Windows сохраняет излишек страниц памяти на жесткий диск, из-за чего виртуальные адреса больше не соответствуют физическим адресам памяти. Попытка процедуры получить доступ к одной из таких страниц генерирует ошибку отсутствия страницы, что указывает Windows на необходимость использовать физическую память. За исключением небольшой задержки, когда страницычитываются с диска обратно в память, ошибки отсутствия страницы обычно не влияют на приложения и сервисы пользовательского режима. Но в режиме ядра при определенных обстоятельствах ошибка отсутствия страницы может вызвать полный сбой системы.

В этом разделе рассматриваются основы программирования в режиме ядра и указываются некоторые наиболее распространенные подводные камни.

## Прерывания и уровни IRQL

Для эффективного реагирования на аппаратные события операционные системы снабжаются механизмом, называемым *прерыванием*. Каждому аппаратному событию назначается прерывание. Например, одно прерывание может инициировать запуск часов и планировщика, другое — драйверы клавиатуры, и т. п. Когда происходит аппаратное событие, Windows доставляет соответствующее прерывание процессору.

Когда процессор получает прерывание, система не создает новый поток для его обслуживания. Вместо этого Windows прерывает существующий поток на короткое время, необходимое для обработчика прерывания об служить прерывание. По завершению обслуживания прерывания система возвращает контроль над потоком его оригинальному владельцу.

Прерывания необязательно инициируются непосредственно аппаратными событиями. Windows также поддерживает программные прерывания в форме процедур отложенного вызова (в дальнейшем — *процедуры DPC* (Deferred Procedure Call)). Windows планирует процедуру DPC, доставляя прерывание соответствующему процессору. Драйверы режима ядра используют процедуры DPC для обработки аспектов аппаратных прерываний, требующих большого расхода времени.

С каждым прерыванием ассоциируется уровень, называемый *уровнем IRQL* (Interrupt ReQuest Level, уровень запроса на прерывание), который определяет большую часть программирования в режиме ядра. Система использует разные значения уровней IRQL, чтобы обеспечить обработку наиболее важных и времязависимых прерываний в первую очередь. По уровню IRQL исполняется процедура обслуживания, назначенная соответствующему прерыванию. Когда происходит прерывание, система находит соответствующую процедуру обслуживания и назначает ее на исполнение процессору.

Текущий уровень IRQL процессора определяет, когда процедура обслуживания запускается на исполнение и может ли она прерывать исполнение потока, исполняемого процессором в настоящее время. В этом отношении основное правило заключается в том, что приоритет имеет наивысший уровень IRQL. Когда процессор получает прерывание, происходит следующее.

- ◆ Если уровень IRQL прерывания выше, чем уровень IRQL процессора, система повышает уровень IRQL процессора до уровня IRQL прерывания.

Исполнение текущего кода на данном процессоре приостанавливается до тех пор, пока не завершится исполнение процедуры обслуживания и уровень IRQL процессора не возвратится к своему первоначальному значению. Исполнение процедуры обслуживания

может быть, в свою очередь, прервано другой процедурой обслуживания, имеющей более высокий уровень IRQL.

- ◆ Если уровень IRQL прерывания такой же, как и уровень IRQL процессора, процедура обслуживания должна ожидать завершения исполнения всех предыдущих процедур с таким же уровнем IRQL.

После этого исполняется процедура текущего прерывания. Ее исполнение может быть прервано прерыванием с еще более высоким уровнем IRQL.

- ◆ Если уровень IRQL прибывшего прерывания ниже текущего уровня IRQL процессора, ее процедура обслуживания должна ожидать завершения исполнения всех предыдущих процедур с более высоким уровнем IRQL.

Эти варианты исполнения процедур обслуживания прерываний с разными уровнями IRQL действительны, когда они исполняются на одном процессоре. Но современные системы обычно оборудованы двумя или более процессорами. Вполне возможна ситуация, когда процедуры драйвера с разными уровнями IRQL исполняются одновременно на разных процессорах. Если процедуры должным образом не синхронизированы, это может вызывать взаимоблокировки.

### Примечание

Уровни IRQL — совсем иное понятие, чем приоритеты потоков. Приоритеты потоков используются системой для управления планированием потоков во время штатной работы системы. Прерывание же по определению является нечто выходящим за рамки штатной работы и требует как можно быстрого обслуживания. Процессор возвращается к нормальной обработке процессов только после завершения обслуживания всех ожидающих выполнения прерываний.

Каждому уровню IRQL назначается численное значение. Но так как эти значения отличаются для разных процессорных архитектур, уровням IRQL обычно присваиваются еще и имена. Драйверы используют только некоторые из всех имеющихся уровней IRQL; остальные уровни IRQL зарезервированы для использования системой. Уровни IRQL, наиболее широко используемые драйверами, приводятся в следующем списке, начиная с уровня с самым низким значением.

- ◆ **PASSIVE\_LEVEL.** Это самый низкий уровень IRQL. Он назначается по умолчанию штатной обработке потоков. Это единственный уровень IRQL, который не ассоциирован с каким-либо прерыванием. Все приложения пользовательского режима исполняются с уровнем IRQL = PASSIVE\_LEVEL так же, как и низкоприоритетные процедуры драйверов. Процедуры, исполняющиеся с уровнем IRQL = PASSIVE\_LEVEL, могут обращаться ко всем сервисам ядра Windows.
- ◆ **DISPATCH\_LEVEL.** Это самый высокий уровень IRQL, назначаемый программным прерываниям. Процедуры DPC и другие процедуры повышенного приоритета исполняются с уровнем IRQL = DISPATCH\_LEVEL. Процедуры, исполняющиеся с уровнем IRQL = DISPATCH\_LEVEL, могут иметь доступ только к ограниченному подмножеству базовых сервисов Windows.
- ◆ **DIRQL.** Этот уровень IRQL выше, чем уровень DISPATCH\_LEVEL, и является самым высшим уровнем IRQL, с которым драйверам обычно приходится иметь дело. В действительности, это собирательный термин для совокупности уровней IRQL, ассоциированных с аппаратными прерываниями, которые также называются *уровнями IRQL устройств* (device IRQL, DIRQL). PnP-менеджер назначает уровень DIRQL каждому уст-

ройству и передает его соответствующему драйверу при запуске. Обычно знать точный уровень не так уж и важно. Для большинства задач нужно лишь знать, что исполнение происходит на уровне DIRQL и что процедура обслуживания блокирует исполнение на данном процессоре практически любой другой задачи. Процедуры, исполняющиеся с уровнем DIRQL, могут иметь доступ только к очень небольшому подмножеству базовых сервисов Windows.

### Внимание!

Одной из основ программирования качественных драйверов является отслеживание уровня IRQL, с которым исполняются или должны исполняться ваши процедуры. Небрежность в этой области может вызвать полный сбой системы. Набор WDK содержит инструменты, такие как, например, Driver Verifier и SDV, которые помогут вам выловить такие ошибки.

Уровни IRQL, с которыми могут исполняться драйверные процедуры, и уровни IRQL, с которых можно вызывать процедуры DDI, описываются в документации WDK. Смотрите эту информацию в разделе **Managing Hardware Priorities** (Управление аппаратными приоритетами) на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79339>.

Дополнительную информацию по уровням IRQL можно найти в *главе 15*.

## Параллелизм и синхронизация

Драйверы обычно многопоточные в том смысле, что они реентерабельные; разные процедуры можно вызывать из разных потоков. Но с драйверами потоки и синхронизация работают существенно иначе, чем с приложениями.

Понятия WDF для потоков и синхронизации представляются в *главе 3*.

### Потоки

Приложения обычно создают и управляют своими собственными потоками. Процедуры же драйверов режима ядра, наоборот, обычно не создают потоков и не исполняются в потоках, созданных специально для них. Драйверы часто работают в большей степени подобно приложениям DLL. Например, если приложение инициирует запрос ввода/вывода, рабочая процедура, которая получает запрос, скорее всего, будет исполняться в потоке, который инициировал запрос, в известном контексте процесса. Но большинство драйверных процедур не "знает" своего контекста процесса и исполняется в произвольном потоке.

Когда процедура исполняется в произвольном потоке, система по существу "одалживает поток", исполняющийся на выделенном процессоре, и использует его для исполнения процедуры драйвера. Драйвер, исполняющийся в произвольном потоке, не может зависеть от каких-либо связей между потоком, в котором он исполняется, и потоком, выдавшим запрос ввода/вывода. Например, если драйвер, получивший запрос на чтение, должен передать его низшему драйверу или обработать прерывание, эти процедуры исполняются в произвольном потоке.

В следующем примере показано, каким образом назначаются потоки в типичном запросе ввода/вывода:

1. Приложение вызывает функцию `DeviceIoControl` в потоке A, чтобы послать драйверу запрос управления устройством.
2. Менеджер ввода/вывода вызывает соответствующую рабочую процедуру, также в потоке A.

3. Драйвер программирует устройство на удовлетворения запроса и возвращает управление приложению.
4. Позже, когда устройство готово, оно создает прерывание, которое обрабатывается соответствующей процедурой ISR (Interrupt Service Routine, процедура обработки прерывания). У драйвера нет возможности проконтролировать, чтобы во время возникновения прерывания на процессоре исполнялся определенный поток, поэтому процедура ISR исполняется в произвольном потоке.
5. Процедура ISR ставит в очередь процедуру DPC, чтобы завершить запрос на более низком уровне IRQL. Процедура DPC также исполняется в произвольном контексте потока.

По случайности процедура DPC может исполняться в контексте потока приложения, которое выдало запрос, но точно так же она может исполняться в потоке совершенно другого процесса. В общем случае, наиболее безопасным будет предположить, что процедура драйвера исполняется в контексте произвольного потока.

Последствия применения контекста произвольного потока могут быть такими.

- ◆ **Разные процедуры обычно исполняются в разных потоках.** В многопроцессорных системах эти две процедуры могли бы прекрасно исполняться одновременно на двух разных процессорах.
- ◆ **Исполнение процедур не должно отбирать больше времени, чем полагается по их уровню IRQL.** Процедура также блокирует всю работу, исполняющуюся на временно захваченном ею потоке. В общем случае, чем выше уровень IRQL, тем более ощутимое воздействие ее исполнения на другие процессы и тем быстрее оно должно завершиться. Процедуры драйверов, исполняющиеся на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , особенно процедуры с уровнем DIRQL, должны избегать таких действий, как длительные остановы или бесконечные циклы, вследствие чего одолженный поток может быть заблокирован бесконечно.

## Синхронизация

Процедуры драйверов могут исполняться параллельно в разных потоках, поэтому во избежание состояния гонок их исполнение необходимо синхронизировать. Например, одновременная попытка двух процедур модифицировать связанный список может нарушить целостность этого списка. Так как в Windows применяется приоритетное планирование, состояние гонок может возникнуть как на однопроцессорной, так и на многопроцессорной системе. Но на многопроцессорной системе возможность возникновения состояния гонок выше, т. к. здесь исполняется одновременно большее число драйверных процедур, чем на однопроцессорной системе.

Принципы синхронизации по существу одинаковые как для режима ядра, так и для пользовательского режима. Но режим ядра представляет дополнительную сложность: доступные методы синхронизации зависят от уровня IRQL.

**Синхронизация процедур уровня PASSIVE\_LEVEL.** Для синхронизации процедур, исполняющихся на уровне  $\text{IRQL} = \text{PASSIVE\_LEVEL}$ , существует несколько средств, включая следующие.

- ◆ **Объекты диспетчера ядра.** Эти объекты — события, семафоры и мьютексы — представляют собой коллекцию объектов синхронизации, широко используемых процедурами, исполняющимися на непроизвольных потоках на уровне  $\text{IRQL} = \text{PASSIVE\_LEVEL}$ .

## Примечание

Кроме синхронизации, события используются и для других целей. Они также применяются в качестве метода сигнализации для таких задач, как обработка завершения ввода/вывода.

- ◆ **Объекты быстрых мьютексов и ресурсов.** Эти объекты создаются поверх объектов диспетчера ядра.

**Синхронизация процедур уровня IRQL = DISPATCH\_LEVEL.** Процедуры драйверов часто исполняются на уровне DISPATCH\_LEVEL и иногда на уровне DIRQL. Основным инструментом для синхронизации процедур уровня DISPATCH\_LEVEL является объект, называемый спин-блокировкой (*spin lock*<sup>1</sup>). Спин-блокировки могут использоваться в контексте произвольного потока на уровне IRQL равном или ниже DISPATCH\_LEVEL. Для защиты ресурса запрашивается и получается спин-блокировка, ресурс используется по назначению, после чего спин-блокировка освобождается. Объект спин-блокировки обычно создается при создании объекта, для защиты которого он предназначается, и откладывается для дальнейшего использования.

Когда процедура получает спин-блокировку, если она уже не исполняется на уровне IRQL = DISPATCH\_LEVEL, она повышается до этого уровня. Когда процедура освобождает блокировку, ее уровень IRQL возвращается в свое прежнее значение.

Процедуры ISR часто необходимо синхронизировать со связанный процедурой DPC, а иногда и с другими процедурами ISR. Но процедуры IRS исполняются на уровне DIRQL, поэтому они не могут использовать обычные спин-блокировки. Вместо этого, процедуры ISR должны использовать спин-блокировки прерываний. Этот объект используется точно таким же образом, как и спин-блокировка, но он повышает уровень IRQL процессора до уровня IRQL процедуры ISR, а не до уровня DISPATCH\_LEVEL.

Эти и связанные с ними методики подробно исследуются в главе 15.

## Состояние гонок и взаимоблокировки

Состояние гонок возникает, когда две или больше процедур пытаются одновременно манипулировать одними и теми же данными. Нераспознанное состояние гонок является распространенной причиной нестабильности драйверов. Но состояние гонок можно предотвратить, управляя условиями, ведущими к его возникновению. Далее приведены два основных подхода.

- ◆ **Синхронизация.** Этот метод использует объекты синхронизации для защиты от доступа разделяемые данные.
- ◆ **Сериализация.** Для предотвращения возникновения состояния гонок этот метод организует в очередь запросы, требующие доступа к разделяемым данным. Таким образом, гарантируется, что эти запросы всегда обрабатываются по порядку и никогда одновременно.

<sup>1</sup> В переводе с английского, одно из значений слова *spin* — вращаться, вертеться. Термин *spin lock* происходит от обстоятельства, что когда один поток владеет спин-блокировкой, все другие потоки, ожидающие получить объект спин-блокировки, вертятся без дела, пока он не освободится. Это можно сравнить с самолетами, кружавшимися над занятым аэродромом, ожидая своей очереди на посадку. — Пер.

### Подсказка

С драйверами временные проблемы могут быть намного более критическими, чем с приложениями, особенно на многопроцессорных системах. Например, на ранних стадиях разработки драйверов разработчики часто используют проверочные сборки как драйвера, так и Windows, которые работают сравнительно медленно. Низкая скорость проверочных сборок может предотвратить перерастание возможного состояния гонок в настоящее. Но когда окончательная версия драйвера исполняется на свободной сборке Windows, вдруг начинают появляться состояния гонок. Методы предотвращения состояния гонок рассматриваются в главе 10.

## Память

Драйверы режима ядра используют память совершенно по-другому, чем процессы пользовательского режима. Оба режима имеют виртуальное адресное пространство, на которое система отображает физические адреса, и оба они используют сходные методики для выделения и управления памятью. Но:

- ◆ компоненты режима ядра разделяют виртуальное адресное пространство;

Это подобно тому, как все приложения DLL, загруженные процессом, разделяют виртуальное адресное пространство этого процесса. В отличие от пользовательского режима, в котором каждый процесс имеет свое виртуальное адресное пространство, разделяемое адресное пространство режима ядра означает, что драйверы режима ядра могут нарушить целостность как памяти своих соседей, так и системной памяти;

- ◆ процессы пользовательского режима не имеют доступа к адресам режима ядра;
- ◆ процессы режима ядра имеют доступ к адресам пользовательского режима, но этот доступ должен осуществляться с осторожностью в корректном контексте приложения.

В режиме ядра указатель на адрес пользовательского режима не имеет строго определенного значения и неправильное обращение с указателем пользовательского режима может создать брешь в безопасности системы или вовсе вызвать ее сбой. Обработка адресов пользовательского режима безопасным образом требует большего, чем простого разыменования указателя.

В главе 8 описывается, как драйверы WDF могут зондировать и блокировать буферы пользовательского режима с целью их безопасной обработки.

## Управление памятью

Драйверы должны эффективно управлять своим использованием памяти. В общем случае, задача управления памятью драйвера во многом такая же, как и задача управления памятью любой другой программы: драйвер должен выделять и освобождать память, использовать память стека должным образом, обрабатывать ошибки отсутствия страницы и т. д. Но среда режима ядра накладывает некоторые дополнительные ограничения.

Ошибки представляют особую проблему для драйверов режима ядра. Ошибка доступа может вызвать аварийный останов bugcheck и последующий системный сбой. Ошибки отсутствия страницы являются особой проблемой, потому что воздействие ошибки отсутствия страницы зависит от уровня IRQL. В частности:

- ◆ в процедурах, исполняющихся на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , ошибка отсутствия страницы вызывает останов bugcheck. На уровне  $\text{IRQL} = \text{DISPATCH\_LEVEL}$  ошибки отсутствия страницы являются распространенной причиной сбоя драйвера;

- ◆ в процедурах, исполняющихся на уровне  $\text{IRQL} < \text{DISPATCH\_LEVEL}$ , ошибка отсутствия страницы обычно не является проблемой. В худшем случае, происходит лишь небольшая задержка, пока страница считывается с диска обратно в память. Но если во время обслуживания ошибки страницы в драйвере происходит другая такая ошибка, драйвер может не иметь достаточно ресурсов для ее обслуживания, по причине использования их для обслуживания предыдущей ошибки. Это создает взаимоблокировку и системный сбой двойной ошибки (double-fault crash).

### Подсказка

Когда при исполнении на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$  в компоненте режима ядра происходит ошибка страницы, код `bugcheck` будет `IRQL_NOT_LESS_OR_EQUAL`. Многие из таких ошибок в исходном коде драйвера можно обнаружить с помощью инструментов PREFast и SDV.

## Пулы памяти

Приложения обычно используют кучу для выделения блоков памяти большого размера или для выделения памяти, которое должно оставаться действительным в течение длительного периода времени. Драйверы режима ядра выделяют память для этих целей, по большому счету, таким же образом. Но из-за проблем, которые ошибки страницы могут вызвать в режиме ядра, драйверы должны использовать две разные кучи — называемые *пулами памяти* — для этого типа выделения памяти.

- ◆ **Страницочный пул.** В случае необходимости, эти страницы памяти могут быть вытеснены из системной памяти на диск.
- ◆ **Нестраницочный пул.** Эти страницы всегда резидентные в памяти и их можно использовать для хранения данных, к которым необходимо получить доступ на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ .

Память в страницочном и нестраницочном пулах часто, соответственно, называют *страницкой* и *нестраницкой* памятью. Память для определенного типа задач необходимо выделять из соответствующего пула. Так, для данных или кода, обращение к которым будет происходить только на уровне  $\text{IRQL} < \text{DISPATCH\_LEVEL}$ , используется страницчная память. Нестраницчная память является дефицитным ресурсом, разделяемым со всеми другими процессами режима ядра, и поэтому ее необходимо бережно использовать.

Вопрос выделения памяти обсуждается в главе 12.

## Стек ядра

Принцип работы стека процедур режима ядра точно такой же, как и стека пользовательского режима. Но максимальный размер стека ядра довольно небольшой, намного меньший, чем максимальный размер стека пользовательского режима. Кроме этого, размер стека режима ядра нельзя увеличивать, поэтому используйте этот ресурс бережно. Вот некоторые советы по использованию стека режима ядра.

- ◆ С рекурсивными процедурами необходимо быть особенно осторожным при использовании стека. В общем, в драйверах нужно избегать использования рекурсивных процедур.
- ◆ Так как путь ввода/вывода диска многоуровневый и часто требует обслуживания одной или двух страницочных ошибок, он очень чувствителен к коэффициенту использования стека.

- ◆ Драйверы, которые обмениваются большими объемами данных, обычно не применяют стек для их хранения.

Вместо этого, драйвер выделяет для объектов нестраничную память и передает указатель на нее. То обстоятельство, что все процессы режима ядра разделяют общее адресное пространство, делает этот подход возможным.

### **Использование стека ядра**

В области управления памятью стек ядра является как вашим лучшим другом, так и опасным врагом. Во многих ситуациях драйвер можно существенно упростить, помещая структуру данных в стек вместо пула. Но так как размер стека ядра очень небольшой, необходимо быть осторожным, чтобы не поместить туда слишком много данных. С драйверами, размещенными внизу очень высоких стеков вызовов (подобно тем, которые обслуживают устройства хранения данных), необходимо быть особенно осторожным, т. к. менеджер памяти и драйверы файловой системы, расположенные выше их, могут занимать существенную часть стекового пространства. (*Питер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

Дополнительную информацию по этому вопросу см. в разделе **How do I keep my driver from running out of kernel-mode stack?** (Как не допустить перерасхода драйвером памяти стека режима ядра?) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79604>.

### **Списки MDL**

Часто драйверы должны передавать или получать от клиентов или устройств данные в буферах. Иногда буфер данных — это просто указатель на область памяти. Но буферы данных режима ядра часто имеют форму списка MDL (Memory Descriptor List, список дескрипторов страниц), который не имеет эквивалента в пользовательском режиме. Список MDL — это структура, описывающая буфер и содержащая список заблокированных страниц в памяти ядра, из которой создан буфер.

Работа со списками MDL рассматривается в главе 8.

## **Советы по программированию в режиме ядра**

Далее следует несколько основных рекомендаций по программированию в режиме ядра.

### **Выделение памяти**

- ◆ **Сделайте драйвер способным обрабатывать ситуации нехватки памяти.** Критически важно избежать сбоя драйверов Windows в ситуациях, когда они не могут выделить память. Если для некоторых аспектов работы драйвера абсолютно необходимо иметь память, выделите ее при инициализации драйвера и сохраните указатель на нее для дальнейшего использования. В противном случае, обработайте неудачную попытку выделения памяти без нарушения работоспособности системы; для этого завершите без успеха связанную операцию или организуйте ее выполнение позже.
- ◆ **Выделяйте тегированную память, чтобы облегчить отладку утечки памяти.** Вопрос выделения памяти обсуждается в главе 12.

## Использование спин-блокировок

- ◆ Не позволяйте процедурам, имеющим спин-блокировку, обращаться к коду или данным в страничной памяти. Даже процедуры, которые обычно исполняются на уровне IRQL = PASSIVE\_LEVEL, исполняются на уровне IRQL = DISPATCH\_LEVEL, когда они удерживают спин-блокировку. Для выявления случаев входа в страничный код с недопустимо высокими приоритетами используйте в этом коде макрос PAGED\_CODE().
- ◆ Не удерживайте спин-блокировки дольше, чем это необходимо. Если процедура A удерживает заблокированный спин-блокировкой ресурс в то время, как процедура B также пытается получить спин-блокировку, то процедура B будет "вращаться" в цикле ожидания до тех пор, пока спин-блокировка не станет доступной. Пока процедура B "вращается" в ожидании, на данном процессоре не может исполняться никакой код с уровнем IRQL = PASSIVE\_LEVEL.
- ◆ Соблюдайте осторожность с получением и освобождением спин-блокировок. Если процедура имеет спин-блокировку и, не освободив ее, пытается получить ее снова, возникает взаимоблокировка.
- ◆ Освобождайте множественные спин-блокировки в порядке, обратном тому, в каком они были получены. Рассмотрим следующий сценарий. Процедура драйвера получает спин-блокировку A, получает спин-блокировку B, освобождает спин-блокировку A и освобождает спин-блокировку B. Этот порядок получения и освобождения спин-блокировок создает интервал, в котором один поток может иметь спин-блокировку A, но не может получить спин-блокировку B, в то время как другой поток имеет спин-блокировку B, но не может получить спин-блокировку A, в результате чего возникает состояние взаимоблокировки.
- ◆ При использовании Driver Verifier включайте опцию обнаружения взаимоблокировок. Функция обнаружения взаимоблокировок инструмента Driver Verifier поможет в верификации и отладке использования спин-блокировок в разрабатываемом коде.

## Управление ошибками страниц

- ◆ Память для определенного типа задач необходимо выделять из соответствующего пула. Определите, из какого пула нужно выделить память: из страничного или нестраничного.
- ◆ При использовании Driver Verifier включайте опцию Force IRQL Checking. Это поможет вам выявить многие проблемы, связанные со страничной памятью.

Использование инструмента Driver Verifier в процессе разработки драйверов излагается в главе 21.

## Обращение к памяти пользовательского режима

- ◆ Не доверяйте ничему, поступающему из пользовательского режима. Всегда проверяйте размер всех получаемых буферов и содержащиеся в них данные на целостность и безопасность. Для прямого ввода/вывода или метода ввода/вывода NEITHER проверяйте достоверность любых параметров, предварительно скопировав их в память ядра, чтобы после проверки приложение не могло изменить их.
- ◆ В процедурах режима ядра не ограничивайтесь простым разыменованием указателей пользовательского режима. В лучшем случае результатом этого будет просто бес-

смыслица, а в худшем вы можете, сами того не ведая, нарушить безопасность системы или же вызвать ее сбой. Чтобы разыменовать указатель пользовательского режима, необходимо, чтобы процедура исполнялась в надлежащем контексте процесса. Также нужно прозондировать и заблокировать память с помощью процедур DDI, которые вызывают исключение в случае нескольких неудачных проверок достоверности адреса. Любую попытку доступа к указателю пользовательского режима необходимо инкапсулировать в блок структурной обработки исключений (например, таких как блок `try/_except`). Таким образом, можно уловить исключения доступа к памяти, которые могут возникнуть, если в процессе обращения к адресу приложение делает его недействительным.

Вопросы, связанные с безопасным доступом к памяти пользовательского режима, рассматриваются в *главе 8*.

## Блокирование потоков

**Соблюдайте осторожность с блокировкой потоков.** Процедура, исполняющаяся с повышенным уровнем IRQL, не позволяет исполняться на данном процессоре никакому другому коду, за исключением кода с еще высшим уровнем IRQL.

Вопрос потоков в режиме ядра рассматривается в *главе 15*.

## Верификация драйверов

- ◆ Пользуйтесь инструментами **Driver Verifier** и **KMDF Verifier**. Начинайте использовать Driver Verifier и KMDF Verifier сразу же после успешной загрузки драйвера. С помощью Driver Verifier можно уловить ошибки, которые трудно обнаружить при нормальной работе, такие как, например, использование кода или данных в странице памяти на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ .
- ◆ Сразу же после компиляции проверяйте код с помощью инструментов **PREFast** и **SDV**. С помощью этих инструментов статического анализа ошибки в коде можно уловить на ранних стадиях разработки, до того, как их станет трудно и сложно устраниТЬ.

Использование инструмента PREFast в процессе разработки драйверов излагается в *главе 23*.

## Использование макросов

- ◆ Применяйте макрос `VERIFY_IS_IRQL_PASSIVE_LEVEL()` в начале всех процедур, которые должны исполняться на уровне `IRQL = PASSIVE_LEVEL`. В драйверах KMDF с помощью макроса `VERIFY_IS_IRQL_PASSIVE_LEVEL()` можно удостовериться в том, что процедура исполняется на уровне `IRQL = PASSIVE_LEVEL`. Если уровень IRQL выше значения `PASSIVE_LEVEL`, макрос генерирует отладочное сообщение. Если отладчик не подключен, макрос `VERIFY_IS_IRQL_PASSIVE_LEVEL()` вызывает исключение. Подобно макросу `VERIFY_IS_IRQL_PASSIVE_LEVEL()`, макрос `PAGED_CODE()` можно использовать в любом драйвере режима ядра. Этот макрос генерирует отладочное сообщение, если процедура исполняется на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ .

Информация по исключениям предоставляется в *главе 15*.

- ◆ Применяйте макрос `UNREFERENCED_PARAMETER` для всех параметров, не используемых процедурой. Этот макрос отключает сообщения компилятора об неиспользуемых па-

метрах. Макрос `UNREFERENCED_PARAMETER` определен в стандартном заголовочном файле `WDK Ntdef.h`, который подключается в файле `Ntddk.h`.

## Основные термины

В этой книге часто употребляется термин "*драйвер*" вместо "*драйвер устройства*", т. к. кроме драйверов устройств имеются и другие типы драйверов. Класс драйверов, называемых программными драйверами, работает по большому счету таким же образом, как и драйверы устройств, только они не управляют устройствами. Наиболее распространенными драйверами в этой категории являются драйверы фильтров.

С целью помочь вам в приобретении базового словарика в области разработки драйверов, далее приводятся определения некоторых терминов, знание которых пригодится вам в понимании дальнейшего материала. Определения дополнительных терминов можно найти в гlosсарии в конце этой книги и в полном гlosсарии в документации набора WDK.

### Devnode

Элемент в дереве устройств PnP-менеджера. Узел `devnode` стека устройства используется PnP-менеджером для хранения информации конфигурации и для отслеживания устройства.

### NTSTATUS

Тип, используемый в качестве возвращаемого значения многими процедурами режима ядра.

### SYS

Расширение имени файла двоичных файлов драйверов режима ядра. Хотя они похожи на файлы DLL, файлы SYS не имеют прямого экспортования.

### Аварийный останов bugcheck

Ошибка, генерируемая, когда целостность структур ядра Windows была безвозвратно нарушена. Иногда также называется *системным сбоем*. Аварийные остановы `bugcheck` генерируются только ошибками при исполнении процессов режима ядра. Когда происходит аварийный останов `bugcheck`, система пытается завершить работу наиболее организованным образом и, в некоторых версиях Windows, выводит сообщение об ошибке на голубом экране. Систему можно также настроить для создания дампа файла останова, который можно потом проанализировать с помощью отладчика ядра.

### Вытесненная страница (paged out)

Страница, временно удаленная из памяти и записанная на диск, пока она снова не понадобится.

### Драйверный пакет (driver package)

Инсталляционный пакет, включающий драйвер и вспомогательные файлы.

### Запрос IOCTL (I/O control, IOCTL)

Тип запроса ввода/вывода, применяющийся для взаимодействия с драйвером для иных целей, чем чтение или запись в устройство. Например, приложение может использовать запрос `IOCTL` для изменения конфигурации устройства или получить номер версии драйвера.

**Интерфейс DDI (device drive interface)**

Коллекция системных процедур, вызываемых драйвером для взаимодействия с сервисами ядра. По существу, интерфейс DDI — это эквивалент интерфейса API для драйверов. Имена процедур интерфейса DDI обычно начинаются префиксами, указывающими их назначение. Например, процедуры KMDF обычно называются `WdfXXX`, а методы UMDF представляются через интерфейс COM, называющийся `IWDFXXX`.

**Менеджер объектов (object manager)**

Сервис ядра, управляющий объектами ядра.

**Нестраничная память (nonpageable memory)**

Память в нестраничном пуле.

**Нестраничный пул (nonpaged pool)**

Куча, которая всегда находится в памяти и никогда не вытесняется на диск.

**Объект диспетчера ядра (kernel dispatcher object)**

Коллекция объектов синхронизации, которые можно использовать при исполнении процедуры на уровне IRQL = PASSIVE\_LEVEL.

**Объект инфраструктуры (framework object)**

Объект, управляемый WDF.

**Объект синхронизации**

Объекты, используемые для защиты от доступа к ресурсам. К таким объектам относятся события (event), семафоры (semaphore), мьютексы (mutex) и спин-блокировки (spin lock).

**Объект устройства (device object)**

Объект устройства, представляющий участие драйвера в обработке запросов ввода/вывода определенного устройства.

**Объект ядра (kernel object)**

Инкапсулированная структура данных, управляемая менеджером объектов ядра.

**Ошибка страницы (page fault)**

Событие, когда процесс пытается обратиться к вытесненной на диск странице памяти.

**Пакет IRP (I/O request packet, IRP)**

Объект, с помощью которого пакет данных и связанной информации передается между менеджером ввода/вывода и компонентами стека устройств. Пакеты IRP также применяются для иных целей, чем выполнение операций ввода/вывода, например, для передачи сообще-

ний от PnP-менеджера. С термином "пакет IRP" тесно связан термин "указатель PIRP", являющийся указателем на пакет IRP. Драйверы WDF обычно не имеют прямых взаимоотношений с пакетами IRP, поэтому в большинстве случаев в этой книге вместо него применяется термин "объект запроса" (request object).

### **Пользовательский режим (user mode)**

Ограниченнный режим работы, в котором исполняются приложения и драйверы UMDF, в котором запрещен прямой доступ к процедурам и структурам данных ядра Windows.

### **Прерывание (interrupt)**

Извещение, посылаемое системе, что произошло что-то вне рамок обычной обработки потока, например, аппаратное событие, которое необходимо обработать как можно скорее.

### **Проверочная сборка (checked build)**

Сборка, применяющаяся только для тестирования и отладки.

### **Произвольный поток (arbitrary thread)**

Поток, исполняющийся процессором в момент, когда система "одолживает" его для исполнения процедуры драйвера, например, такой как процедура ISR. Процедура драйвера не знает, какой процесс является владельцем потока.

### **Процедура DPC (Deferred Procedure Call)**

Процедура, которую код, исполняемый на уровне DIRQL, может отложить и выполнить позже на уровне DISPATCH\_LEVEL.

### **Процедура ISR (Interrupt Service Routine, ISR)**

Процедура, реализуемая драйвером устройства для обработки аппаратных прерываний.

### **Процедура завершения ввода/вывода (I/O completion routine)**

Процедура драйвера, исполняющаяся по завершению запроса ввода/вывода расположенным ниже драйвером в стеке.

### **Процедура обслуживания (service routine)**

Процедура, выполняющая обработку для прерывания.

### **Пул памяти (memory pool)**

То же самое, что и куча.

### **Рабочий элемент (work item)**

Механизм, применяемый процедурами уровня High-IRQL для частичного исполнения на уровне IRQL = PASSIVE LEVEL.

**Режим ядра (kernel mode)**

Режим, в котором процессы имеют такие самопривилегии и риски, как и ядро операционной системы Windows.

**Свободная сборка (free build)**

Сборка для окончательного продукта.

**Спин-блокировка (spin lock)**

Объект синхронизации, который можно использовать при исполнении на уровне IRQL = DISPATCH\_LEVEL.

**Спин-блокировка прерывания (interrupt spin lock)**

Объект синхронизации, который можно использовать при исполнении на уровне DIRQL.

**Стек устройства (device stack)**

Коллекция объектов устройств и связанных драйверов, которые обрабатывают взаимодействие с определенным устройством.

**Страницчная память (pageable memory)**

Память в страницном пуле.

**Страницочный пул (paged pool)**

Куча, которую при необходимости можно записать на диск и, когда понадобится, считать обратно в память.

**Уровень IRQL (Interrupt Request Level, IRQL)**

Численное значение, которое Windows назначает каждому прерыванию. В случае конфликта прерывание с высшим уровнем IRQL имеет приоритет, и его процедура обслуживания исполняется первой.

**Файл INF (INF)**

Текстовый файл, содержащий данные по установке драйвера.

# ГЛАВА 3

## Основы WDF

Драйверная модель WDF определяет объектно-ориентированную, событийно-управляемую среду как для драйверов режима ядра (KMDF), так и для драйверов пользовательского режима (UMDF). Код драйвера управляет функциями, специфичными для устройства, а инфраструктура от Microsoft вызывает драйвер WDF в ответ на события, которые влияют на работу его устройства.

В этой главе представлены основные концепции конструкции и реализации инфраструктуры WDF для драйверов UMDF и KMDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Документация WDK</b>	
User-Mode Driver Framework Design Guide <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79341">http://go.microsoft.com/fwlink/?LinkId=79341</a>
Kernel-Mode Driver Framework Design Guide <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79342">http://go.microsoft.com/fwlink/?LinkId=79342</a>
<b>Прочее</b>	
Writing Secure Code (авторы — Howard и LeBlanc) <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80091">http://go.microsoft.com/fwlink/?LinkId=80091</a>

## WDF и WDM

Драйверы WDF используются для таких же целей, как и драйверы WDM: они осуществляют взаимодействие между Windows и устройством. Хотя WDF представляет совершенно новую драйверную модель, она не является отдельной от WDM. WDF выполняет функцию слоя абстракции между моделью WDM и WDF-драйвером, который упрощает задачу создания надежных, безопасных и эффективных драйверов.

WDF предоставляет инфраструктуру, которая выполняет ключевые задачи драйверов WDM: получает и обрабатывает пакеты IRP, управляет изменениями в состоянии Plug and Play и энергопотребления, и т. д. А для предоставления функциональности, специфичной для уст-

<sup>1</sup> Справочник по организации инфраструктуры драйверов пользовательского режима. — *Пер.*

<sup>2</sup> Справочник по организации инфраструктуры драйверов режима ядра. — *Пер.*

<sup>3</sup> Говард, Лебланк. Как создавать безопасный код. — *Пер.*

ройства, инфраструктура вызывает драйвер WDF клиента. Хотя WDF поддерживает две инфраструктуры — UMDF и KMDF — высокуюровневая конструкция и функциональность обеих этих инфраструктур сходны.

В этой главе дается концептуальный обзор WDF и драйверов WDF, при этом упор делается на базовые возможности, общие для обеих инфраструктур WDF. А в *главе 4* обсуждается, каким образом реализуются эти две инфраструктуры и типы устройств, поддерживаемые ими.

## Что такое WDF?

WDF предоставляет общую драйверную модель для широкого круга типов устройств. Наиболее важными особенностями модели являются следующие.

- ◆ Поддержка драйверов, как пользовательского режима, так и режима ядра. Базовая часть модели применима к драйверам обеих типов.
- ◆ Хорошо спроектированная объектная модель. Драйверы WDF взаимодействуют с объектами посредством надежного и стойкого интерфейса.
- ◆ Объектная иерархия, которая упрощает управление временем жизни объектов и синхронизацию запросов ввода/вывода.
- ◆ Модель ввода/вывода, в которой инфраструктуры управляют взаимодействиями с операционной системой. Инфраструктуры также управляют потоком запросов ввода/вывода, включая реагирование на события Plug and Play и энергопотребления.
- ◆ Реализация механизма Plug and Play и управления энергопотреблением, которая предоставляет надежное управление состояниями и стандартное интеллектуальное управление переходами между состояниями. Драйверы WDF обрабатывают только те переходы между состояниями, которые имеют отношение к их устройству.

Инфраструктура предлагает набор объектов, которые представляют различные, связанные с Windows и драйверами, базовые структурные компоненты, такие как устройство, драйвер, запрос ввода/вывода, очередь и т. д. Драйверы WDF используют объекты инфраструктуры для реализации различных аспектов функциональности драйвера. Например, если драйверу WDF необходима очередь для управления запросами ввода/вывода, он создает объект очереди инфраструктуры.

Каждый объект представляет программный интерфейс, который драйвер использует для доступа к свойствам объекта или чтобы дать объекту указания выполнять задачи. Каждый объект также поддерживает одно или более событий, которые вызываются в ответ на такие явления, как изменение системного состояния или прибытие запроса ввода/вывода. События позволяют инфраструктуре известить драйвер WDF о том, что произошло что-то, требующее его внимания, и передать управление соответствующему драйверу WDF для обработки этого происшествия. Например, объект очереди инфраструктуры создает событие, чтобы известить драйвер WDF о том, что запрос ввода/вывода готов для обработки.

Некоторые объекты инфраструктуры создаются самой инфраструктурой и передаются драйверу WDF. Другие объекты инфраструктуры создаются по мере того, как у драйвера WDF возникает в них потребность. Ваша задача, как разработчика, состоит в том, чтобы собрать соответствующие объекты в структуру, поддерживающую требования устройства. Эта структура следит, каким образом инфраструктура направляет ввод/вывод через драйвер WDF и каким образом этот драйвер взаимодействует с операционной системой и устройст-

вом. В зависимости от конкретных требований устройства, структуры отличаются в подробностях реализации, но все драйверы WDF создаются из одних и тех же базовых блоков.

Некоторые объекты инфраструктуры постоянны или имеют очень долгое время жизни. Например, при физическом подключении устройства типичный драйвер WDF создает объект устройства для представления устройства и один или несколько объектов очереди для управления запросами ввода/вывода. Эти объекты обычно существуют на протяжении времени жизни данного устройства. Другие же объекты недолговечны; они служат для решения определенных кратковременных задач и уничтожаются сразу же после решения этой задачи. Например, драйвер WDF может создать объект запроса ввода/вывода, чтобы издать один запрос ввода/вывода, а после завершения запроса уничтожает этот объект.

По умолчанию инфраструктура предоставляет обработку для всех событий, поэтому драйверам WDF не обязательно обрабатывать их. Но в результате этой полной зависимости от инфраструктуры для стандартной обработки событий получается функциональный, но относительно неинтересный драйвер. Чтобы предоставить необходимую специфичную для устройства поддержку, драйверы WDF должны обрабатывать, по крайней мере, некоторые события.

Чтобы подменить стандартную обработку события инфраструктурой своей обработкой, драйвер WDF должен явным образом зарегистрировать специальную функцию обратного вызова. Инфраструктура вызывает эту функцию обратного вызова, чтобы известить драйвер WDF о возникшем событии и позволить ему выполнить необходимое обслуживание. Эта модель опционной обработки событий драйвером является ключевым аспектом WDF. Инфраструктура предоставляет стандартную интеллектуальную обработку для всех событий; если стандартной обработки достаточно для события, драйверу WDF не нужно регистрировать функцию обратного вызова. Драйверы WDF регистрируют функции обратного вызова только для тех событий, для которых стандартная обработка, предоставляемая инфраструктурой, не является достаточной. Все другие события обрабатываются инфраструктурой, без вмешательства драйвера.

## Объектная модель WDF

Объектная модель WDF определяет набор объектов, представляющих общеупотребляемые структурные компоненты драйверов, такие как устройства, память, очереди, запросы ввода/вывода и сам драйвер. Объекты инфраструктуры имеют четко определенные жизненные циклы и контракты, и драйвер WDF взаимодействует с ними посредством хорошо структурированных интерфейсов. Объекты UMDF реализуются в виде объектов COM, а объекты KMDF — в виде комбинации непрозрачных "дескрипторов" и функций, которые работают с этими дескрипторами. Но на концептуальном уровне объектные модели WDF для инфраструктур KMDF и UMDF похожи друг на друга.

Объекты инфраструктуры создаются как самой инфраструктурой, так и клиентским драйвером WDF. Кто создает объект, инфраструктура или драйвер, зависит от конкретного объекта и его предполагаемого использования:

- ◆ объекты файлов создаются самой инфраструктурой и передаются драйверу WDF;
- ◆ объекты устройств должны создаваться драйвером WDF;
- ◆ объекты запросов ввода/вывода и объекты памяти, в зависимости от обстоятельств, могут создаваться либо инфраструктурой, либо драйвером WDF.

## Программный интерфейс

Объект инфраструктуры предоставляет программный интерфейс, состоящий из свойств, методов и событий.

- ◆ *Свойства* представляют характеристики объекта. С каждым свойством связан метод для получения значения свойства и, если применимо, отдельный метод для установки его значения.
- ◆ *Методы* выполняют какие-либо действия над объектом или указывают объекту на необходимость выполнять действия.
- ◆ *События* — это связанные с объектом происшествия, на которые драйвер WDF может реагировать, такие как, например, прибытие запроса ввода/вывода или изменение в состоянии энергопотребления.

Для обработки события драйвер WDF создает обратный вызов и регистрирует его в инфраструктуре:

- ◆ драйвер UMDF создает объекты обратного вызова на основе COM, которые предоставляют интерфейсы обратного вызова для обработки событий;
- ◆ драйвер KMDF создает функции обратного вызова.

Независимо от того, каким образом они создаются, оба типа обратных вызовов работают одинаково. Так как инфраструктура предоставляет стандартные обработчики для всех событий, драйверы WDF регистрируют обратные вызовы только для тех событий, которые имеют отношение к их устройству.

При возникновении события:

- ◆ если драйвер WDF зарегистрировал обратный вызов для события, инфраструктура активирует его и драйвер обрабатывает событие;
- ◆ если драйвер WDF не зарегистрировал обратный вызов для события, инфраструктура активирует стандартный обработчик события.

## Иерархия объектов

Объекты инфраструктуры организованы в иерархию, которую инфраструктуры используют для управления такими аспектами, как время жизни объектов, очистка объектов и область синхронизации. Например, если объект имеет потомков, при удалении объекта всего его потомки также удаляются. Иерархия основана не на взаимоотношениях наследования между различными объектами инфраструктуры, а на области видимости различных объектов и порядке, в котором их требуется уничтожить. Иерархия определяется следующим образом:

- ◆ в корне иерархии находится объект драйвера инфраструктуры; все остальные объекты являются его потомками;
- ◆ некоторые объекты, такие как объекты очереди, всегда должны быть потомками объекта устройства или объекта, который является потомком объекта устройства;
- ◆ некоторые объекты, например объекты памяти, могут иметь одного или несколько родителей.

Объектная модель WDF в деталях рассматривается в главе 5.

## Параллелизм и синхронизация

Управление параллельными операциями является проблемной задачей для большинства программ. WDF упрощает эту задачу с помощью нескольких внутренних механизмов синхронизации. В частности, когда инфраструктура активирует обратный вызов, драйвер может указать ей применить блокировку. Иерархия объектов WDF поддерживает возможность, называемую *областью синхронизации* (которая также называется границей блокировки), которая позволяет драйверу WDF указать, какой объект нужно заблокировать, когда инфраструктура вызывает обратные вызовы события ввода/вывода драйвера.

Вопросы синхронизации в общем и область синхронизации в частности обсуждаются в *главе 10*.

### Совет

Для разработки драйверов KMDF необходимо знать механизмы уровней IRQL, а также синхронизации и блокировки режима ядра. Эти вопросы рассматриваются в *главе 15*.

## Модель ввода/вывода

Запрос ввода/вывода, посыпаемый Windows драйверу WDF, получает инфраструктура, которая и занимается механикой планирования, постановки в очередь, завершения и отмены выполнения запросов от имени драйверов. По прибытию запроса ввода/вывода инфраструктура определяет, должна ли она обработать запрос сама или предоставить это драйверу WDF, активировав обратный вызов. Если обрабатывать запрос будет драйвер WDF, инфраструктура упаковывает данные в объект запроса инфраструктуры и передает его драйверу.

Инфраструктура отслеживает каждый запрос ввода/вывода. Так как инфраструктура осведомлена обо всех активных запросах, она может активировать соответствующие обратные вызовы при отмене запроса ввода/вывода, изменениях в состоянии энергопотребления, удалении устройства и т. п.

Драйвер WDF управляет потоком запросов ввода/вывода, создавая один или несколько объектов очереди и конфигурируя каждый из них в соответствии с:

- ◆ типом запроса ввода/вывода, который обрабатывается очередью;
- ◆ способом отправки запросов из очереди;
- ◆ воздействием событий управления энергопотреблением на очередь.

Для получения запросов WDF-драйверы регистрируют обратные вызовы очереди, а объект очереди отправляет запросы, активируя соответствующий обратный вызов. Драйвер WDF может сконфигурировать объект очереди для отправки запросов одним из следующих способов.

- ◆ **Параллельно.** Объект очереди отправляет запросы драйверу, сразу же по их прибытию. Сразу несколько запросов могут быть активными одновременно.
- ◆ **Последовательно.** Объект очереди отправляет запросы драйверу, но не отправляет новый запрос до тех пор, пока предыдущий не был завершен или переслан другой очереди.
- ◆ **Вручную.** Драйвер извлекает запросы из очереди по мере необходимости.

Plug and Play и события управления энергопотреблением могут воздействовать на состояние очередей запросов ввода/вывода. Инфраструктура предоставляет интегрированную под-

держку Plug and Play и управления энергопотребление для очередей запросов ввода/вывода, и она совмещает организацию очередей с отменой сообщений. Драйвер WDF может сконфигурировать очередь таким образом, чтобы инфраструктура запускала, останавливалась или возобновляла процесс организации очереди в ответ на события Plug and Play или энергопотребления. Драйверы WDF могут явным образом начинать, останавливать и возобновлять процесс организации очередей, а также очищать очереди.

## Отмена запросов ввода/вывода

Так как по существу ввод/вывод Windows является асинхронным, отмену запросов ввода/вывода часто трудно обрабатывать должным образом. Драйвер должен принимать во внимание возможные состояния гонок и предупредить их возникновение, для чего может потребоваться одна или больше блокировок. Драйверы WDM могут самостоятельно управлять блокировками, и необходимый для этого код обычно разбросан среди нескольких драйверных процедур. Инфраструктура предоставляет стандартную обработку отмены запросов ввода/вывода посредством управления блокировками для запросов ввода/вывода и через отмену поставленных в очередь запросов, не требуя для этого вмешательства драйвера. Драйверы WDF, использующие стандартные установки WDF, обычно не требуют больших (или совсем никаких) процедур для выполнения отмены.

С применением WDF отмена запроса ввода/вывода происходит при следующих обстоятельствах.

- ◆ По умолчанию инфраструктура управляет отменой запросов, которые находятся в очереди.
- ◆ Запросы, которые были извлечены из очереди и отправлены драйверу WDF, нельзя отменить, если только они не были явно помечены драйвером, что их можно отменять.
- ◆ Драйверы WDF могут указывать, имеет ли право инфраструктура отменять запросы, находящиеся в активной стадии обработки драйвером. Эта возможность позволяет драйверу WDF без трудностей поддерживать отмену долго исполняющихся запросов. Инфраструктура помогает драйверу WDF управлять свойственными процессу отмены состояниями гонок, а драйвер в основном отвечает за предоставление необходимого кода для отмены запроса.

Вопросы, связанные с вводом/выводом, обсуждаются в *главе 8*.

## Исполнители ввода/вывода

Иногда драйверы WDF должны отсылать запросы ввода/вывода другим драйверам. Например:

- ◆ некоторые драйверы WDF могут обрабатывать все запросы ввода/вывода самостоятельно, но многим драйверам требуется передать, по крайней мере, некоторые полученные ими запросы для обслуживания вниз по стеку;
- ◆ некоторые драйверы WDF должны инициировать запросы ввода/вывода. Например, функциональные драйверы иногда посыпают запросы IOCTL вниз по стеку, чтобы получить информацию от драйвера шины;
- ◆ некоторые драйверы WDF должны отсылать запросы ввода/вывода совершенно другому стеку устройства. Например, драйвер может нуждаться в информации от другого стека устройства, прежде чем он может выполнить запрос.

Драйверы WDF посылают запросы исполнителю ввода/вывода, который является объектом инфраструктуры, представляющий драйвер, который должен получить запрос. Для драйвера WDF исполнителем ввода/вывода по умолчанию является нижележащий драйвер в стеке. Но исполнитель ввода/вывода также может представлять другой драйвер в том же стеке или драйвер в совершенно другом стеке. Драйверы WDF могут посылать запросы исполнителю ввода/вывода синхронно или асинхронно. Также они могут указывать значение тайм-аут для любого типа запроса, таким образом, информируя инфраструктуру, сколько времени ей отводится на завершение запроса, по истечению которого его можно отменить.

Объекты исполнителей ввода/вывода поддерживают программный интерфейс, используемый драйверами WDF для отслеживания состояния исполнителя, создания запросов в формате, требуемом исполнителем, получения информации об исполнителе, а также получения извещения об удалении исполнителя. Объекты исполнителей ввода/вывода также отслеживают находящиеся в очереди и отосланные запросы, и они могут отменять ожидающие выполнения запросы в случае изменений в состоянии устройства исполнителя или в состоянии драйвера WDF, приславшего запрос.

Вопросы, связанные с исполнителями ввода/вывода, обсуждаются в *главе 9*.

## Обработка некритических ошибок

Драйверы WDF вызывают методы WDF для выполнения разных задач. Многие из этих вызовов функций могут завершиться неудачно по следующим причинам.

- ◆ Иногда ошибка настолько серьезная, что драйвер не может восстановить свою работоспособность. В результате таких критических ошибок в UMDF происходит сбой главного процесса, а в KMDF — останов bugcheck.
- ◆ Некоторые ошибки менее серьезны и не влияют на работу устройства или драйвера в такой степени, что драйвер не может восстановить свою работоспособность. В таком случае, функция докладывает о характере ошибки драйверу, возвращая соответствующее значение состояния. Для драйверов UMDF это будет HRESULT, а для драйверов KMDF — NTSTATUS. Драйвер потом обрабатывает ошибку должным образом.

Тема критических ошибок рассматривается в *главе 22*.

Необходимо проявлять скрупулезность в проверке возвращаемых значений для ошибок, с тем, чтобы обрабатывать их должным образом. Но неудачно завершиться могут только функции WDF, возвращающие значение статуса. Все другие функции гарантируют просто возврат значения соответствующего типа, хотя в некоторых случаях это возвращаемое значение может быть NULL.

Некоторые драйверы WDF обнаруживают ошибки сами. Но только те обратные вызовы, которые возвращают значение статуса, должны заботиться о возвращении ошибок. В таком случае, обратный вызов докладывает о некритических ошибках инфраструктуре, возвращая соответствующее значение статуса.

### Совет

Программа статического анализа PREfast выявляет все случаи невыполнения проверки возвращаемого статуса. Подробности приводятся в *главе 23*.

## Извещение об ошибках UMDF

Тип HRESULT поддерживает несколько кодов успешного и неуспешного завершения.

- ◆ Драйверы UMDF могут проверять значения HRESULT, возвращаемые методами UMDF для состояний ошибок, передавая возвращаемое значение макросу `SUCCEEDED` или `FAILED`. Если ситуация этого требует, драйвер может исследовать сам код ошибки, чтобы принять решение о дальнейших действиях.
- ◆ Обратные вызовы драйверов UMDF при ошибках должны возвращать соответствующее значение HRESULT.

Значения HRESULT, которые инфраструктура готова получить, перечисляются на справочной странице для каждого метода WDK.

Подробно значения HRESULT обсуждаются в *главе 18*.

## Извещение об ошибках KMDF

Тип NTSTATUS также поддерживает несколько кодов успешного и неуспешного завершения.

- ◆ Драйверы KMDF могут проверить значения NTSTATUS, возвращенные функциями KMDF, передавая возвращенное значение макросу `NT_SUCCESS`. Если ситуация этого требует, драйвер может исследовать сам код ошибки, чтобы принять решение о дальнейших действиях.
- ◆ Обратные вызовы драйверов KMDF при ошибках должны возвращать соответствующее значение NTSTATUS.

Значения NTSTATUS, которые инфраструктура ожидает получить, перечисляются на справочной странице для каждого метода WDK.

### Примечание

Определенные в Windows значения NTSTATUS находятся в файле Ntstatus.h, входящем в WDK. Можно также определить специальные значения NTSTATUS для обработки обстоятельств, не охваченных значениями в файле Ntstatus.h. В общем случае, специальные значения применяются только в тех обстоятельствах, когда можно ожидать, что оба компонента понимают смысл значения. Для передачи информации о состоянии между драйвером KMDF и инфраструктурой применяется несколько специальных значений NTSTATUS. Если запрос ввода/вывода завершается специальным KMDF-значением NTSTATUS, инфраструктура KMDF преобразовывает специальный код ошибки в хорошо известное значение NTSTATUS, которое потом возвращается создателю запроса.

Специальные значения NTSTATUS инфраструктуры KMDF определяются в файле %wdk%\inc\wdf\kmdf\НомерВерсии\Wdfstatus.h.

## Plug and Play и управление энергопотреблением

Прямая и надежная обработка событий Plug and Play и энергопотребления является критически важной для обеспечения надежности системы и получения пользователем положительного впечатления от работы с ней. Главной целью в разработке WDF было упростить реализацию поддержки Plug and Play и управления энергопотреблением и сделать эту поддержку доступной как драйверам UMDF, так и драйверам KMDF.

В драйверах WDF нет надобности в реализации сложной логики, необходимой для отслеживания состояния Plug and Play и энергопотребления. Внутренне инфраструктура поддерживает набор машин состояния, которые управляют механизмом Plug and Play и состоянием энергопотребления для драйверов WDF. Инфраструктура извещает драйверы WDF об изменениях в состоянии Plug and Play и энергопотребления посредством последовательности событий, каждому из которых прямо сопоставлены специфичные для устройства действия, которые драйвер может выполнять.

## **Критерии конструкции WDF для Plug and Play и управления энергопотреблением**

Поддержка WDF для Plug and Play и управления энергопотреблением основана на следующих принципах.

- ◆ Драйвер не обязан интерпретировать или отвечать на каждое событие Plug and Play или управления энергопотреблением. Вместо этого, драйвер должен иметь возможность подключиться к обработке и обрабатывать только те события, которые имеют отношение к его устройству, оставляя обработку других событий инфраструктуре.
- ◆ Действия WDF на каждом этапе должны быть четко определенными и предсказуемыми. В сущности, к каждому событию Plug and Play и энергопотребления применяется контракт, в котором четко определены ответственности драйвера.
- ◆ События Plug and Play и управления энергопотреблением должны быть основаны на группе основных изменений состояния — подключении питания и отключении питания. Для обработки конкретных сценариев к этой основной группе необходимо добавить действия. Например, действие остановов механизма Plug and Play активирует основное событие выключения питания плюс событие освобождения аппаратного обеспечения.
- ◆ События Plug and Play и управления энергопотреблением должны быть связаны с четко определенной задачей. Драйверы не должны отслеживать состояние системы, чтобы выяснить, каким образом отвечать на определенное событие.
- ◆ Инфраструктуры должны предоставить стандартное поведение для обширного набора возможностей Plug and Play и управления энергопотреблением, включая перераспределение ресурсов, останов устройств, удаление устройств, извлечение устройств, быстрое возобновление работы, отключение простаивающих устройств и активацию устройств внешними сигналами.
- ◆ Драйвер должен обладать способностью заменять любые предоставляемые инфраструктурой стандартные функции своими.
- ◆ Механизм Plug and Play и управление энергопотреблением должны быть полностью интегрированы с другими компонентами инфраструктуры, такими как, например, управление потоком запросов ввода/вывода.
- ◆ Инфраструктура должна поддерживать оборудование и драйверы как простой, так и сложной конструкции. Реализация простых задач должна быть легкой, но инфраструктура должна предоставить разработчикам простой способ для наращивания возможностей своей конструкции для обработки сложных задач.
- ◆ Инфраструктура должна быть расширяемой и гибкой в соответствующих точках, с тем, чтобы драйверы могли реагировать на изменения состояний посредством своих собственных действий. Например, WDF поддерживает самоуправляемый ввод/вывод, с по-

мощью которого можно координировать операции, не связанные с поставленными в очередь запросами ввода/вывода, или операции, не связанные с управлением энергопотреблением таких изменений состояния.

Инфраструктура предоставляет стандартную обработку для каждого события Plug and Play и энергопотребления. Этот подход чрезвычайно уменьшает количество решений, которые драйвер WDF должен принимать, особенно при переходах между состояниями энергопотребления. Драйверы WDF содержат значительно меньший объем кода, связанного с Plug and Play и управлением энергопотреблением, чем драйверы WDM. Некоторые драйверы WDF не требуют кода для работы с Plug and Play и управления энергопотреблением вообще.

Подробная информация о том, каким образом WDF обрабатывает изменения Plug and Play и состояния энергопотребления, приводится в *главе 7*.

## Безопасность

Каждый драйвер должен быть безопасным. Пользователи доверяют драйверам с передачей данных между своими приложениями и устройствами. Небезопасный драйвер может подвергнуть опасности кражи конфиденциальные данные пользователя, такие как, например, пароли или имена учетных записей. Небезопасные драйверы также могут сделать пользователей уязвимыми в других ситуациях, таких как, например, атаки DoS (Denial of Service, отказ в обслуживании) или спуфинг.

Драйверы представляют большую опасность, чем приложения, по следующим причинам:

- ◆ обычно доступ к драйверам может получить любой пользователь, а также группы пользователей;
- ◆ пользователи обычно не знают, что они пользуются драйвером.

Вопрос обеспечения безопасности тесно связан с обеспечением надежности. Хотя эти два требования иногда требуют решения разных вопросов программирования, небезопасный драйвер не может быть надежным, и наоборот. Например, драйвер, вызывающий системный сбой, по существу осуществляет атаку DoS.

Описание всех аспектов безопасности драйверов выходит за рамки данной книги; для этого потребовалось бы написать отдельную книгу. Но знание нескольких базовых понятий в этой области может существенно помочь в создании более безопасных и надежных драйверов. Центральное правило безопасности драйверов WDF гласит, что каждый драйвер должен работать таким образом, чтобы не допустить несанкционированного доступа к устройствам, файлам и данным. В конструкцию WDF заложена функциональность содействовать обеспечению безопасности драйверов, предоставляя стандартные безопасные установки и выполнения всестороннюю проверку параметров.

В этом разделе вкратце рассматриваются средства обеспечения безопасности WDF. Дополнительную информацию по этому вопросу можно найти в документации WDK, которая предоставляет базовые руководящие принципы, применимые как к драйверам UMDF, так и к драйверам KMDF.

Кроме этого, руководящие принципы по безопасности доступны в разделе **Creating Reliable and Secure Drivers** (Создание надежных и безопасных драйверов) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80063>. А в книге "Writing Secure Code, Second Edition" (авторы Говард и Лебланк, <http://go.microsoft.com/fwlink/?LinkId=80091>) рассматриваются общие методы написания безопасного кода.

## Стандартные безопасные установки

Если только драйвер WDF не указывает другое, инфраструктура требует привилегии администратора для доступа к конфиденциальной информации, такой как предоставляемые имена, идентификаторы устройств или интерфейсы управления WMI. Стандартные установки безопасности WDF предоставляют безопасные имена для объектов устройств режима ядра, с тем, чтобы только компоненты режима ядра и администраторы могли иметь доступ к этим объектам.

Вопрос безопасности объектов устройств рассматривается в *главе 6*.

Инфраструктура автоматически предоставляет стандартную обработку для всех необрабатываемых драйвером запросов ввода/вывода. Стандартная обработка предотвращает вход неожиданных запросов в драйвер WDF, который может неправильно их интерпретировать. Но WDF не защищает драйвер от ошибочно сформированных запросов. Когда драйверу WDF приходится обрабатывать такие запросы, ответственность за их правильную обработку лежит на нем.

По умолчанию драйверы UMDF исполняются с учетной записью LocalService, обладающей минимальными привилегиями безопасности. Но когда клиентское приложение открывает драйвер UMDF, оно может предоставить драйверу разрешение имперсонировать (т. е. выдавать себе за) данное приложение. Это позволяет драйверу выполнять задачи, требующие повышенного уровня привилегий.

Имперсонация обсуждается в *главе 8*.

## Проверка достоверности параметров

Крупным источником проблем безопасности, как в драйверах, так и в приложениях, является переполнение буфера. Все функции и методы WDF, которые используют буфера, имеют параметр длины, указывающий требуемый минимальный объем буфера. Для запросов ввода/вывода эта практика WDF расширена, и буфера ввода/вывода помещаются в объекты памяти инфраструктур. Предоставляемые объектами памяти методы доступа к данным автоматически проверяют достоверность объема буфера и определяют, позволяют ли указанные для буфера разрешения запись в память, находящуюся ниже в стеке.

Одна из наиболее распространенных проблем безопасности происходит, когда драйвер режима ядра не обрабатывает должным образом буфера пользовательского режима, содержащиеся в запросах IOCTL. Это особенно относится к запросам, указывающим METHOD\_NEITHER-тип ввода/вывода. Так как в этом типе запросов указатели на буфера пользовательского режима по своей природе небезопасны, по умолчанию драйверы KMDF не имеют к ним прямого доступа. Вместо этого, WDF предоставляет методы, чтобы позволить драйверу WDF безопасно проверить буфер и заблокировать его в памяти.

## Поддержка верификации, трассировки и отладки

WDF содержит несколько возможностей для тестирования и трассировки, чтобы облегчить выявление проблем на ранних стадиях цикла разработки. Это такие возможности, как:

- ◆ встроенная проверка верификатором инфраструктур;
- ◆ встроенная регистрация трассировки;
- ◆ расширения отладчика.

WDF содержит верификаторы для обеих инфраструктур для поддержки специфичных для инфраструктур возможностей верификации, которых в настоящее время нет в инструменте Driver Verifier (Verifier.exe). Верификаторы WDF издают исчерпывающие сообщения трассировки, предоставляющие подробную информацию о протекающих процессах внутри самой инфраструктуры. Они также отслеживают ссылки на все объекты WDF и создают журнал результатов трассировки, который можно просмотреть с помощью WinDbg.

Тема трассировки рассматривается в *главе 11*.

WDF поддерживает два набора расширений отладчика для WinDbg: один для отладки драйверов UMDF, а второй — драйверов KMDF. Эти расширения дополняют базовые возможности отладчика WinDbg, предоставляя подробную информацию о специфичных для WDF проблемах.

Тема отладки рассматривается в *главе 22*.

## Обслуживаемость и управление версиями

Распространенной проблемой для драйверов является обслуживаемость. Когда компания Microsoft выпускает новую версию Windows, поставщики драйверов должны протестировать свои драйверы и убедиться в том, что они работают должным образом с новой версией операционной системы. С любым драйвером, в котором используются недокументированные возможности, или документированные возможности используются нестандартным образом, скорее всего, будут проблемы с совместимостью при переходе на следующую версию Windows.

Драйверы WDF менее подвержены таким проблемам, т. к. компания Microsoft тестирует инфраструктуры с каждой новой версией Windows. Это тестирование позволяет удостовериться в том, что драйверы WDF ведут себя последовательным образом от одной к следующей версии Windows.

Чтобы улучшить обслуживаемость драйвера и способствовать предотвращению проблем совместимости, WDF предоставляет средства управления версиями и поддержку для параллельной работы разных основных версий инфраструктуры. Каждый выпуск инфраструктуры имеет основную и дополнительную версии. Драйвер WDF всегда использует основную версию выпуска, для которого он был создан и протестирован. Если устанавливается более новая основная версия инфраструктуры, предыдущая версия продолжает работать параллельно с ней, и драйверы WDF продолжают привязываться к той основной версии, для которой они были разработаны.

При компиляции драйверы WDF указывают основную и дополнительную версии инфраструктуры, для которой они были созданы. При установке драйвера он указывает версию инфраструктуры, которая должна присутствовать в системе, чтобы удовлетворить минимальные требования драйвера. При этом возможны следующие сценарии.

- ◆ Если указанная драйвером основная версия не установлена, то соинсталлятор WDF, который включен во все установочные пакеты WDF, автоматически устанавливает необходимую основную версию параллельно с любыми установленными более старыми основными версиями.
- ◆ Если указанная дополнительная версия более новая, чем та, которая установлена в системе, соинсталлятор WDF обновляет инфраструктуру до уровня более новой дополнительной версии.

Более старая дополнительная версия сохраняется как резервная копия, которая может быть установлена обратно с помощью функции восстановления системы в случае необходимости.

При установке новой дополнительной версии драйверы WDF, использующие основную версию того же выпуска, автоматически привязываются к этой более новой дополнительной версии. Это означает, что драйвер WDF привязывается к более новой основной версии, под которую драйвер был скомпилирован. Таким образом, драйверы WDF могут пользоваться исправлениями ошибок, выполненных в дополнительных версиях.

### Совет

Незначительные изменения в версиях Windows могут повлиять даже на драйверы, которые соответствуют требованиям WDF. Поэтому, чтобы удостовериться в отсутствии проблем в разрабатываемых драйверах, всегда тестируйте их с вариантами пакетов обновлений и новыми версиями Windows. Установка драйверов и управление версиями рассматриваются в главе 20.

### Почему я решил работать над WDF?

Во время разработки Windows 2000 (приблизительно в 1998 году), когда я учился, как создавать драйверы и отлаживать код режима ядра, я постоянно поражался тому, что вся система была работоспособной. Я видел в драйвере три разных уровня сложности: первый — во взаимодействии со своей аппаратурой; второй — в реализации Plug and Play и управлении энергопотреблением и других системных требований; и третий — во взаимодействии с другими драйверами.

Когда я думал о том, как трудно было правильно реализовать только одну из этих задач, не говоря уже о реализации всех трех и управлении взаимодействием между ними, у меня просто кружилась голова. Даже в то далекое время я знал, что должен быть лучший способ разработки драйверов, но у меня не было достаточно опыта и времени попробовать найти этот лучший способ. Так что, когда я услышал о команде разработчиков драйверных инфраструктур, я знал, что я должен работать в этой команде, с тем чтобы решить эти вопросы и сделать задачу разработки драйверов более простой. (Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation*, Microsoft.)

# **ЧАСТЬ II**

## **Изучение инфраструктур**

**Глава 4.** Обзор драйверных инфраструктур

**Глава 5.** Объектная модель WDF

**Глава 6.** Структура драйвера и его инициализация

## ГЛАВА 4

# Обзор драйверных инфраструктур

WDF состоит из двух инфраструктур: UMDF, для поддержки драйверов пользовательского режима, и KMDF, для поддержки драйверов режима ядра. Хотя обе инфраструктуры имеют общую базовую конструкцию, детали реализации каждой инфраструктуры и соответствующих драйверов отличаются существенным образом. Имея выбор из двух инфраструктур, разработчикам драйверов приходится решать, какую из них использовать для реализации драйвера для конкретного устройства. В этой главе рассматривается, каким образом реализованы эти две инфраструктуры, как работают драйверы внутри каждой из них, а также типы устройств, поддерживаемых каждой инфраструктурой. Также рассматриваются некоторые руководящие принципы для избрания наилучшего решения для каждого типа устройства.

Ресурсы, необходимые для данной главы	Расположение
Документация WDK	
Введение в WDF	<a href="http://go.microsoft.com/fwlink/?LinkId=82316">http://go.microsoft.com/fwlink/?LinkId=82316</a>
Приступление к работе с драйверной инфраструктурой режима ядра	<a href="http://go.microsoft.com/fwlink/?LinkId=82317">http://go.microsoft.com/fwlink/?LinkId=82317</a>

## Обзор драйверных инфраструктур

Хотя обе инфраструктуры содержат несколько возможностей, которые поддерживают только специфичные требования драйверов UMDF или KMDF, большинство базовых возможностей WDF общие для обеих инфраструктур.

- ◆ Инфраструктуры предоставляют объекты, которые являются абстракциями стандартных объектов, требуемых драйверами, такие как объекты драйверов, объекты устройств и объекты запросов ввода/вывода.
- ◆ Драйверы обращаются к объектам инфраструктур посредством единообразного хорошо спроектированного интерфейса.
- ◆ Инфраструктуры управляют временем жизни объектов, отслеживая используемые объекты и освобождая их, когда они больше не нужны.
- ◆ Инфраструктуры реализуют общие драйверные возможности, такие как машины состояний для управления Plug and Play и энергопотреблением, синхронизацией, очередями ввода/вывода, доступом к реестру и отменой запросов ввода/вывода.

- ◆ Инфраструктуры также управляют потоком запросов ввода/вывода от Windows к драйверу и координируют очереди запросов ввода/вывода с извещениями Plug and Play и управления энергопотреблением.
- ◆ Каждая инфраструктура работает единообразно во всех поддерживаемых версиях Windows, так что драйверы WDF могут исполняться на новой версии Windows, не требуя никаких или очень незначительных модификаций.

Ключевым принципом WDF является требование, что код драйвера должен применяться только для управления устройством, а не для обработки рутинных взаимодействий с системой, таких как отслеживание состояния Plug and Play и энергопотребления. WDF предоставляет слой абстракции между драйвером и Windows, так что для большинства сервисов драйверы взаимодействуют с инфраструктурами. Эта модель позволяет WDF предоставлять интеллектуальные, тщательно проверенные решения для распространенных операций. Драйвер WDF может вступить в действие для того, чтобы заменить представляемую WDF стандартную обработку своей обработкой, необходимой для удовлетворения специфических требований своего устройства, после чего позволяет инфраструктуре выполнить оставшиеся операции. Например, разработчикам не требуется писать большой объем рутинного кода лишь для того, чтобы успешно установить и загрузить драйвер.

Инфраструктура исполняет роль посредника во взаимодействиях между драйвером WDF и другими драйверами и между драйвером WDF и Windows. Драйвер WDF взаимодействует с Windows и другими драйверами посредством различных объектов инфраструктуры, которые применяются для таких целей, как работа с запросами ввода/вывода, управление потоком ввода/вывода, синхронизация, буферизация памяти, а также обращение к аппаратным ресурсам, таким как регистры устройств или прерывания.

Кроме возможностей поддержки, используемых всеми категориями устройств, WDF поддерживает расширения для определенных категорий устройств, с тем, чтобы предоставить возможности поддержки, специфичные этим типам устройств. Например, как UMDF, так и KMDF предоставляют расширения для поддержания связи с устройствами USB. При добавлении в Windows новых возможностей и предоставлении поддержки новым категориям устройств, возможности, общие для всех поддерживаемых категорий устройств, будут добавляться в интерфейс DDI инфраструктур.

В следующих двух разделах рассматриваются возможности, специфичные для UMDF- и KMDF-инфраструктур.

## Обзор инфраструктуры UMDF

Инфраструктура UMDF — это программная модель на основе COM, которая поддерживает разработку драйверов функций и фильтров для протокольных устройств Plug and Plug, таких как, например, устройства USB. Инфраструктура представляет собой библиотеку DDL, которая реализует объектную модель WDF, представляя внутрипроцессные, основанные на COM, объекты драйверов, устройств, запросов ввода/вывода и т. п. Драйвер — это файл DLL, состоящий из набора внутрипроцессных объектов обратного вызова на основе COM, которые обрабатывают события, вызванные соответствующими объектами инфраструктуры.

Технология COM была избрана в качестве основы для инфраструктуры UMDF по следующим причинам:

- ◆ многие программисты уже знакомы с этой технологией;
- ◆ интерфейсы COM позволяют логическое группирование функций, что способствует пониманию и передвижению по интерфейсу DDI с легкостью;

- ◆ существуют многочисленные инструменты, включая библиотеку ATL (Active Template Library, библиотека шаблонных классов), поддерживающие приложения и объекты на основе COM.

Информация о программировании COM приводится в главе 18.

Программная модель UMDF основана на следующих двух типах объектов.

- ◆ **Объекты инфраструктуры.** Эти объекты принадлежат инфраструктуре и представляют сам драйвер, устройство, управляемое драйвером, очереди ввода/вывода для управления запросами ввода/вывода и т. д.
- ◆ **Объекты функций обратного вызова.** Эти объекты создаются драйвером, который регистрирует их в UMDF. Каждый объект предоставляет специфичные для устройства ответные действия на события, создаваемые объектом инфраструктуры.

Интерфейс DDI инфраструктуры UMDF, который включает как объекты, предоставляемые инфраструктурой, так и объекты функций обратного вызова, создаваемые драйвером, предоставляется полностью посредством интерфейсов COM. Для взаимодействия с объектами инфраструктуры, драйверы применяют указатели интерфейса, а не указатели объектов. В свою очередь, объекты инфраструктуры взаимодействуют с создаваемыми драйверами объектами обратного вызова посредством соответствующего интерфейса.

## Объекты инфраструктуры UMDF

Объекты инфраструктуры UMDF могут создаваться следующими способами.

- ◆ Инфраструктурой. Например, когда инфраструктура загружает драйвер, она создает инфраструктурный объект драйвера и передает объекту обратного вызова драйвера указатель на интерфейсы `IWDFDriver` и `IWDFDeviceInitialize` объекта драйвера.
- ◆ Драйвером. Например, драйвер создает инфраструктурный объект устройства, вызывая метод `IWDFDriver::CreateDevice`.
- ◆ В зависимости от обстоятельств, некоторые объекты, такие как, например, объекты запросов ввода/вывода, могут создаваться либо инфраструктурой, либо драйвером.

Интерфейсам, которые предоставляются инфраструктурными объектами, присваиваются имена в формате `IWDOBJECT`, где `Object` означает объект, который экспортирует данный интерфейс. Кроме интерфейса `IUnknown`, все инфраструктурные объекты предоставляют один или больше интерфейсов, которые позволяют обращаться к функциональным возможностям объекта. Например, инфраструктурный объект устройства также экспортирует интерфейсы `IWDFDevice` и `IWDOBJECT`. Каждый интерфейс имеет один или несколько методов и свойств для выполнения следующих задач:

- ◆ методы указывают объекту выполнить какое-либо действие;
- ◆ свойства предоставляют доступ к данным.

Обычно каждое свойство состоит из двух методов: один метод используется для чтения существующего значения, а второй — для установки нового значения. Свойства, не позволяющие выполнять чтение или запись, имеют только один метод. Для разных типов методов применяются разные форматы имен.

Правила именования методов UMDF обсуждаются в главе 5.

## Объекты обратного вызова инфраструктуры UMDF

Инфраструктура предоставляет стандартную обработку для всех событий. Драйверы могут подменять стандартную обработку событий инфраструктурой своей обработкой, создавая объекты обратного вызова для обработки событий, требующих специального внимания, и регистрируя эти объекты в инфраструктуре. Объекты обратного вызова предоставляют один или несколько интерфейсов обратного вызова, каждый из которых обрабатывает набор связанных событий. Интерфейсы обратного вызова состоят из одного или нескольких методов (по одному методу для каждого события), которые инфраструктура вызывает, чтобы известить драйвер о событии и передать объекту обратного вызова любые связанные данные.

Драйверы регистрируют объекты обратного вызова, передавая указатель на один из интерфейсов объекта инфраструктурному объекту, создавшему связанное событие. В зависимости от того, как создается инфраструктурный объект — драйвером или инфраструктурой, драйверы регистрируют объекты обратного вызова одним из двух способов.

- ◆ **Инфраструктурный объект создается драйвером.** Драйвер обычно регистрирует объекты обратного вызова, когда он вызывает инфраструктурный метод, который создает инфраструктурный объект. Например, драйвер регистрирует свой объект обратного вызова, когда он вызывает метод `IWDFDriver::CreateDevice` для создания инфраструктурного объекта устройства.
- ◆ **Инфраструктурный объект создается инфраструктурой.** Объекты, создаваемые инфраструктурой, обычно предоставляют один или несколько методов, которые драйвер может вызвать для регистрации объекта обратного вызова для определенного события. Например, чтобы зарегистрировать объект обратного вызова завершения запроса для запроса ввода/вывода, драйвер вызывает метод `IWDFToRequest::SetCompletionCallback` объекта запроса ввода/вывода.

События, которые драйвер обрабатывает, определяют, какие интерфейсы предоставляются каждым объектом обратного вызова. Например, если драйвер должен обрабатывать событие чтения объекта очереди, но не его событие очистки, объект обратного вызова предоставляет только один интерфейс обратного вызова — `IQueueCallbackRead`. Для обработки как события чтения, так и события очистки, объект обратного вызова предоставил бы интерфейсы `IQueueCallbackRead` и `IObjectCleanup`. Иногда инфраструктурный объект не генерирует никаких особых событий; в таком случае в объекте обратного вызова нет надобности. Например, часто драйверам не требуется обрабатывать события, генерируемые объектами памяти.

### Примечание

Объекты обратного вызова не обязательно ограничены обработкой событий, генерируемых только одним инфраструктурным объектом. Например, объект обратного вызова образца устройства `Fx2_Driver` предоставляет три интерфейса обратного вызова: `IPnpCallbackHardware`, `IPnpCallback` и `IRequestCallbackRequestCompletion`. Первые два интерфейса регистрируются в инфраструктурном объекте устройства для обработки событий управления Plug and Play и энергопотребления. А третий интерфейс, `IRequestCallbackRequestCompletion`, регистрируется в инфраструктурном объекте очереди для обработки завершения запроса.

## Обзор инфраструктуры KMDF

Инфраструктура KMDF представляет собой программную модель, которая поддерживает разработку функциональных драйверов, драйверов фильтра и драйверов шины. Сама инфра-

структура — это разделяемая множественными драйверами реентерабельная библиотека. Драйверы динамически привязываются к библиотеке во время процесса загрузки. Система может поддерживать несколько параллельных версий инфраструктуры, и каждая основная версия может поддерживать несколько драйверов.

Инфраструктура обрабатывает базовые задачи драйвера WDM и вызывает сопоставленный драйвер KMDF, предоставляемый поставщиком оборудования, для выполнения задач, специфичных для устройства. С помощью KMDF можно разрабатывать следующие типы драйверов:

- ◆ функциональные драйверы для устройств Plug and Play;
- ◆ драйверы фильтра для устройств Plug and Play;
- ◆ драйверы шины для стеков устройств Plug and Play;
- ◆ некоторые типы драйверов мини-портов, включая NDIS-WDM, протокол NDIS и Smartcard;
- ◆ драйверы, не поддерживающие Plug and Play, для унаследованных устройств, не являющихся частью стека Plug and Play, например, подобные драйверам для Windows NT 4.0.

Драйвер вызывает инфраструктурный метод во время создания стека устройства, чтобы предоставить инфраструктуре информацию о типе устройства. Кроме этого, функциональные драйверы, драйверы фильтров и шин реализуют различные наборы функций обратного вызова. Например, драйвер шины обычно поддерживает обратные вызовы для перечисления потомков устройства шины. Функциональный драйвер обычно поддерживает обратные вызовы для обработки запросов ввода/вывода и состояния энергопотребления своего устройства.

Интерфейс DDI инфраструктуры KMDF поддерживает все типы драйверов. Но некоторые части интерфейса DDI специально предназначены для функциональных драйверов и драйверов фильтров и шин. Кроме этого, в зависимости от типа драйвера, стандартная обработка некоторых событий инфраструктурой KMDF может быть разной. Это особенно касается обработки запросов ввода/вывода. Например, если драйвер фильтра не обрабатывает запрос, инфраструктура автоматически передает запрос следующему нижележащему драйверу в стеке. А если функциональный драйвер не обрабатывает запрос, инфраструктура завершает запрос неудачей.

## Объекты KMDF

В отличие от объектов обратного вызова, применяемых драйверами UMDF, драйверы KMDF реализуют функции обратного вызова. Объекты инфраструктуры KMDF могут создаваться следующими способами.

- ◆ Инфраструктура создает объект и передает его дескриптор драйверу.

Например, при получении запроса ввода/вывода инфраструктура создает объект запроса ввода/вывода и передает его дескриптор процедуре обратного вызова *EvtIoXXX* драйвера.

- ◆ Драйвер создает объект, вызывая инфраструктурный метод создания объекта.

Например, драйвер создает инфраструктурный объект устройства, вызывая метод *WdfDeviceCreate*.

- ◆ В зависимости от обстоятельств, некоторые объекты, такие как объекты запросов ввода/вывода, могут создаваться либо инфраструктурой, либо драйвером.

Инфраструктурные объекты предоставляют функции двух типов:

- ◆ методы указывают объекту выполнять какое-либо действие;
- ◆ свойства предоставляют доступ к данным.

Обычно каждое свойство состоит из двух методов: один метод используется для чтения существующего значения, а второй — для установки нового значения. Свойства, не позволяющие выполнять чтение или запись, имеют только один метод. Для разных типов методов применяются разные форматы имен.

Правила именования методов KMDF обсуждаются в главе 5.

### **От эволюционного развития к революционному**

Вначале целью KMDF было усовершенствовать существующую драйверную модель эволюционным образом, решая проблему по частям. Что касается меня лично, два аспекта сместили цели инфраструктуры в направлении революционного изменения модели. Первым аспектом было определение конкретной объектной модели. Объектная модель коренным образом изменила способ очистки и освобождения ресурсов, т. к. она предоставила четкий контракт касательно методов взаимодействия объектов друг с другом и с самим драйвером, освобождая драйвер от необходимости отслеживать состояние и удалять объекты вручную.

Вторым аспектом было введение машин состояний для Plug and Play, управления энергопотреблением и политики энергопотребления. Эти машины состояний позволили переместить процесс создания почти всех объектов — особенно объектов очередей ввода/вывода, объектов исполнителей ввода/вывода и объектов устройств — в инфраструктуру. С перемещением всего управления состоянием в инфраструктуру, код драйвера уменьшился до объема, необходимого лишь для работы с оборудованием, а не со всей окружающей его операционной системой.

Подобно объектной модели, машины состояний также определили очень четкий контракт на поведение инфраструктуры и драйверов. Вместе, объектная модель и машины состояний дали толчок к развитию инфраструктуры не в последовательности небольших шагов в течение времени, а сразу же одним большим скачком. (Даун Холэн (*Down Holan*), команда разработчиков Windows Driver Foundation, Microsoft.)

## **Функции обратного вызова инфраструктуры KMDF**

Если драйверу требуется обработать какое-либо событие, которое может быть сгенерировано инфраструктурным объектом, он должен создать соответствующую функцию обратного вызова и зарегистрировать ее в инфраструктуре. В WDK функции обратного вызова именуются в формате *EvtObjectEvent*, где *Object* обозначает соответствующий инфраструктурный объект, а *Event* — соответствующее событие. Например, устройство переходит в состояние энергопотребления D0 (рабочий режим), инфраструктура вызывает функцию обратного вызова драйвера, называемую *EvtDeviceD0Entr*. Но этот формат — всего лишь принятное в документации соглашение, которое не является обязательным. Функциям обратного вызова по событию драйверов KMDF можно присваивать любые имена.

В главе 5 приводятся рекомендации по именованию функций обратного вызова инфраструктуры KMDF.

## **Архитектура WDF**

Архитектура WDF разрабатывалась с тем, чтобы инфраструктуры играли роль посредника в большинстве взаимодействий между драйвером и Windows и между драйвером и его уст-

ройством. В этом разделе рассматривается архитектура WDF и взаимодействие ее основных компонентов с драйвером при обработке запросов ввода/вывода и при выполнении других основных задач. В следующих двух разделах приводится обобщенное описание, каким образом эта архитектура реализована для драйверов UMDF и KMDF.

Концептуальное представление архитектуры WDF показано на рис. 4.1.

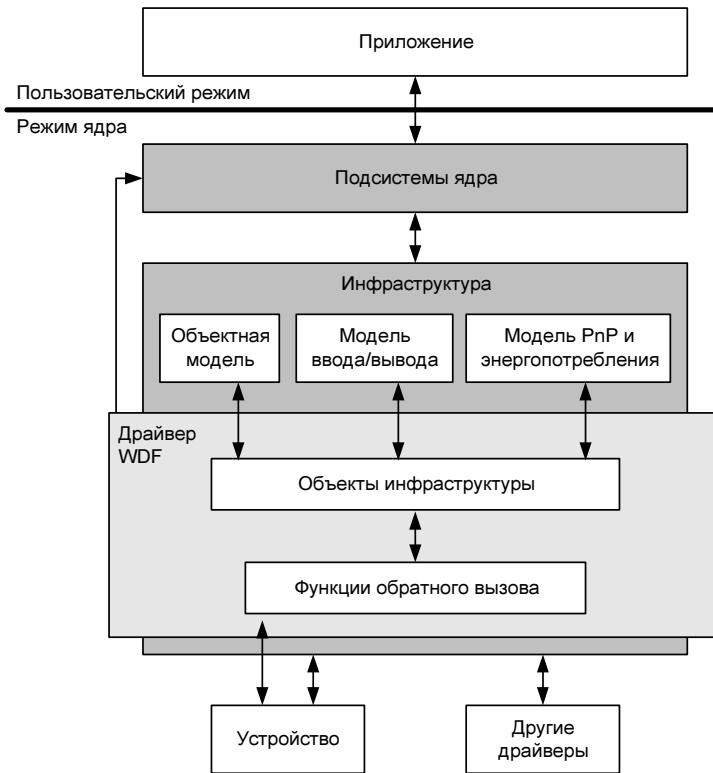


Рис. 4.1. Концептуальное представление модели WDF

**Приложения.** Приложения обычно взаимодействуют с драйверами WDF точно так же, как они бы взаимодействовали с драйверами WDM. Приложение получает дескриптор устройства обычным способом и вызывает соответствующие функции Windows API, чтобы послать запросы устройству. В большинстве случаев приложение не знает или не имеет надобности знать, с кем оно взаимодействует, с драйвером WDM или с драйвером WDF.

### Приложения и драйверы UMDF

Одной из наших главных целей при разработке инфраструктуры UMDF было решение, чтобы приложение не могло различать драйвер пользовательского режима и драйвер режима ядра. Это большое преимущество перед некоторыми другими моделями классовых устройств пользовательского режима, которые применимы только к определенному типу устройства и которые требуют модификации приложений, прежде чем они могут использовать драйвер.

Нам не удалось получить повсеместную прозрачность. Например, мы разрешаем все три режима передачи для запросов IOCTL, но мы не можем предоставить драйверу доступ к адресному пространству клиента, поэтому некоторые аспекты ввода/вывода режимов

DIRECT и NEITHER недоступны. Мы добавили новый код статуса, который может быть возвращен после обработки операции ввода/вывода, чтобы сообщить приложению, что процесс драйвера был прекращен во время обработки ввода/вывода. Чтобы разрешить имперсонацию, приложение должно вызывать по-иному функцию `CreateFile` для имперсонации пользователя драйвером. Но за исключением вышеперечисленного, с точки зрения приложения, драйвер UMDF должен быть неотличим от драйвера KMDF. (*Питер Вилендт (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

**Подсистемы ядра.** Это те же самые подсистемы — менеджер ввода/вывода, менеджер PnP и т. д., которые осуществляют взаимодействие с драйверами WDM. С точки зрения этих подсистем, инфраструктура является драйвером WDM, который представлен объектом устройства WDM. Сервисы получают запросы от приложений и передают их в пакетах IRP инфраструктурам. По завершению запроса сервисы возвращают результаты приложению.

**Верхняя кромка инфраструктуры.** Верхняя кромка инфраструктуры служит в качестве слоя абстракции между Windows и драйвером. Драйвер может получить большую часть необходимых ему сервисов Windows, вызывая DLL инфраструктуры. Но если драйверу требуются сервисы, которые интерфейс DDI инфраструктуры не предоставляет, он может вызывать функции Windows.

Инфраструктуры содержат следующие три ключевые модели, которые определяют, каким образом обрабатывается большая часть задач драйвера.

- ◆ *Объектная модель* определяет, каким образом создаются и управляются объекты инфраструктуры.
- ◆ *Модель ввода/вывода* определяет, каким образом обрабатываются запросы ввода/вывода. В случае с запросами ввода/вывода на чтение, запись и IOCTL, инфраструктура получает от Windows пакет IRP, переупаковывает данные в формате объектной модели и пересыпает их драйверу WDF для обработки. По окончанию обработки драйвер возвращает запрос инфраструктуре. По завершению запроса инфраструктура завершает пакет IRP и возвращает результаты менеджеру ввода/вывода, который, в конечном счете, возвращает результаты приложению.
- ◆ *Модель Plug and Play и энергопотребления* определяет, каким образом управляются изменения состояния Plug and Play и энергопотребления.

В драйверах WDF нет надобности в реализации сложной машины состояний, необходимой для отслеживания состояния Plug and Play и энергопотребления и определения необходимых корректировок при изменениях в состоянии. Для запросов управления Plug and Play и энергопотреблением машины состояний инфраструктуры определяют необходимое действие, исходя из конкретного полученного пакета IRP Plug and Play или энергопотребления и текущего состояния системы. Модель содержит набор событий, соотнесенных непосредственно со специфичными для устройства действиями. Драйверы регистрируют обратные вызовы только для тех событий, которые имеют отношение к их устройству, и оставляют обработку других событий инфраструктурам.

**Объекты инфраструктуры.** Основными строительными блоками драйверов WDF являются объекты инфраструктуры. Они представляют общеупотребляемые структурные компоненты драйверов, такие как устройства, память, очереди ввода/вывода и т. д. Драйвер взаимодействует с этими объектами посредством четко определенных интерфейсов. Эти объекты создаются либо инфраструктурой и передаются драйверу, либо драйвером с помощью методов, предоставляемых инфраструктурой.

**Обратные вызовы драйверов.** В случае событий, требующих их участия в обработке, драйверы заменяют предоставляемую инфраструктурой обработку по умолчанию своей обработкой. Чтобы подменить стандартную обработку события, предоставляемую инфраструктурой, своей обработкой, драйвер реализует и регистрирует обратный вызов. Когда происходит событие, инфраструктура активирует обратный вызов и передает драйверу любые связанные данные. Драйвер обрабатывает событие и возвращает результаты инфраструктуре.

### **Когда применять обратные вызовы**

Мне всегда нравится говорить, что стандартная обработка событий основной инфраструктурой WDF уже сама собой является полностью функциональным драйвером. Это стандартное поведение нужно подменять только тогда, когда вы хотите выполнить какую-либо специальную обработку события. (*Питер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

**Нижняя кромка инфраструктуры.** Драйверам часто требуется взаимодействовать со своим устройством или другими драйверами. Рассмотрим примеры.

- ◆ Драйвер WDF взаимодействует с другими драйверами с помощью объекта инфраструктуры, который называется объектом исполнителя ввода/вывода.

Механизм передачи запросов другим драйверам и возвращения результатов драйверу WDF воплощается инфраструктурой.

- ◆ Для управления обмена данными с устройствами, поддерживающими DMA, драйверы KMDF используют объекты DMA инфраструктуры.

Драйверы UMDF не взаимодействуют непосредственно с устройствами. Вместо этого, запросы передаются инфраструктурой соответствующему драйверу режима ядра, который и обрабатывает передачу данных.

## **Инфраструктура UMDF**

С точки зрения менеджера ввода/вывода Windows, исполнителем запроса ввода/вывода должен быть объект устройства WDM. Эти объекты находятся в адресном пространстве режима ядра и недоступны драйверам пользовательского режима. Ключевым требованием для UMDF является предоставление безопасного способа перехода из пользовательского режима в режим ядра и доставки запросов ввода/вывода драйверу пользовательского режима. Это требование удовлетворяется следующими основными элементами инфраструктуры UMDF:

- ◆ драйвером режима ядра, называемым отражателем (reflector), который представляет драйверы UMDF в части режима ядра стека устройства. Отражатель управляет взаимодействием с хост-процессом UMDF, который исполняется в пользовательском режиме;
- ◆ механизмом взаимодействия, который позволяет передавать запросы через границу между пользовательским режимом и режимом ядра хост-процессу и в обратном направлении.

На рис. 4.2 представлена схема инфраструктуры UMDF и соответствующего драйвера. Большая часть компонентов, показанных в схеме, предоставляется компанией Microsoft. Основным исключением является драйвер (или драйверы) UMDF и иногда нижний стек устройства, которые могут быть реализованы поставщиком аппаратуры.

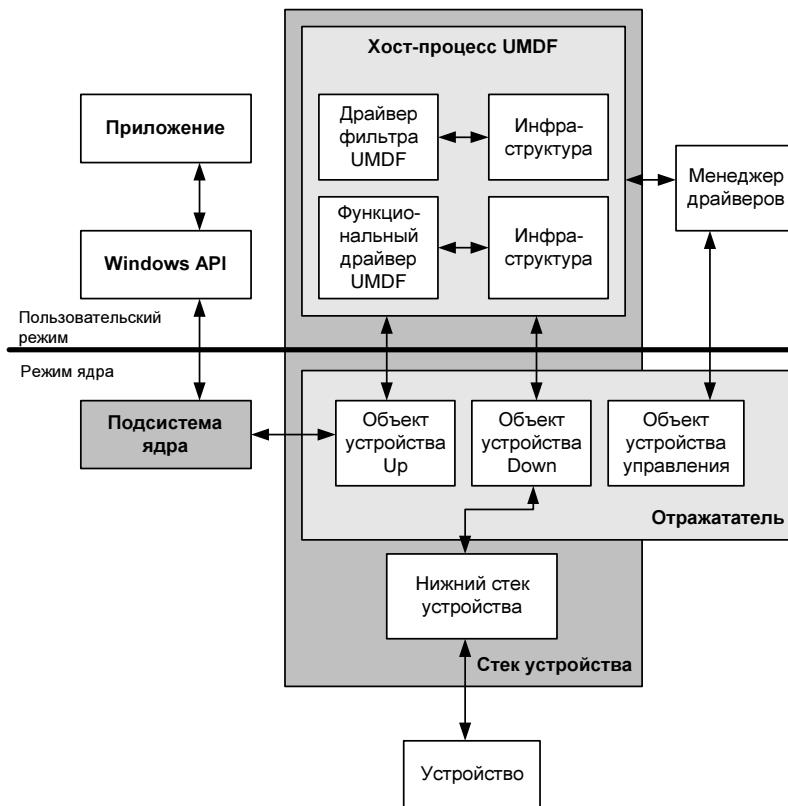


Рис. 4.2. Схема инфраструктуры и драйвера UMDF

## Компоненты инфраструктуры UMDF

Каждый драйвер UMDF функционирует как часть стека устройства, который управляет устройством. Часть стека исполняется в пользовательском режиме, а часть — в режиме ядра. Делается это следующим образом.

- ◆ **Верхняя часть стека исполняется в пользовательском режиме.** Она состоит из хост-процесса, называемого Wudfhost, который служит для загрузки драйвера, и DLL инфраструктуры. Хост-процесс также принимает участие в управлении взаимодействием между драйверами в стеке UMDF и между компонентами стека пользовательского режима и режима ядра.
- ◆ **Нижняя часть стека исполняется в режиме ядра.** Эта часть заведует фактическим взаимодействием с подсистемами ядра и устройством. В некоторых случаях, например с драйверами USB, нижний стек устройства предоставляется компанией Microsoft. В других случаях он реализуется поставщиком аппаратуры.

### Совет

С некоторыми драйверами UMDF единственным назначением части стека устройства, исполняющейся в режиме ядра, является добавление этого устройства в дерево устройств менеджера PnP. Примерами таких драйверов могут служить драйверы чисто программных

устройств и драйверы подключенных через сеть устройств, которые взаимодействуют со своими устройствами посредством Windows API.

В оставшемся материале этого раздела дается краткое описание компонентов, составляющих инфраструктуру UMDF (см. рис. 4.2).

**Хост-процесс UMDF (host process).** Хост-процесс Wudfhost содержит компоненты, составляющие часть стека устройства, исполняющегося в пользовательском режиме, и служит посредником во взаимодействии между этими компонентами и отражателем. Каждый стек UMDF имеет собственный хост-процесс. Основными компонентами хост-процесса являются следующие.

- ◆ **Стек UMDF.** Этот стек состоит из одного или нескольких функциональных драйверов или драйверов фильтра. Стек UMDF имеет свои объекты устройств и пакеты IRP, связанные с лежащими ниже структурами данных WDM. Стек UMDF обрабатывает запросы ввода/вывода в основном таким же образом, как это делает стек режима ядра, хотя существуют отличия в деталях.
- ◆ **Один или несколько экземпляров инфраструктуры.** Для каждого драйвера UMDF в стеке имеется один экземпляр инфраструктуры. Инфраструктура представляет собой библиотеку, которая поддерживает объектную модель UMDF, и служит посредником во взаимодействии между драйверами и хост-процессом.

Хост-процесс Wudfhost является потомком процесса менеджера драйверов. Хотя он не является сервисом Windows, хост-процесс Wudfhost исполняется с параметрами доступа учетной записи LocalService.

Для получения дополнительной информации об учетной записи LocalService и других учетных записях см. раздел **Service User Accounts** в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=82318>.

### **WDF, UMDF и WUDF**

Иногда мы применяем сокращение WUDF, иногда UMDF, а иногда KMDF. Это просто WDF. Вся эта неразбериха с сокращениями является побочным эффектом запуска проектов KMDF и UMDF. Первоначально было лишь одно сокращение WDF — Windows Driver Framework (драйверная инфраструктура Windows). Потом применялся термин Windows User-Mode Driver Framework (драйверная инфраструктура Windows пользовательского режима). Некоторое время также использовался термин Windows Kernel-Mode Driver Framework (драйверная инфраструктура Windows режима ядра). Наконец мы решили использовать сокращения KMDF и UMDF, но к тому времени старые сокращения были уже "засементированы" в базовом коде. (Питер Виландт (*Peter Wielandt*), команда разработчиков *Windows Driver Foundation*, Microsoft.)

**Менеджер драйверов (Driver Manager).** Менеджер драйверов — это сервис Windows, который управляет всеми хост-процессами UMDF в системе. Он создает хост-процесс, отслеживает состояние во время исполнения хост-процессов и останавливает хост-процессы, когда они больше не нужны. Если нет загруженных драйверов, менеджер драйверов отключен, и включается после загрузки первого UMDF-драйвера.

**Отражатель (Reflector).** Отражатель — это драйвер режима ядра, который поддерживает все драйверы UMDF в системе. Он управляет взаимодействием между компонентами режима ядра, такими как стеки устройств режима ядра, и хост-процессами UMDF. Рефлектор создает следующие два объекта устройства для каждого стека устройства UMDF в системе.

- ◆ **Объект устройства Up (Up Device Object).** Этот объект устанавливается вверху стека устройства режима ядра для устройства. Он принимает запросы от менеджера вво-

да/вывода, передает их хост-процессу UMDF для обработки и после их завершения возвращает менеджеру ввода/вывода.

- ◆ **Объект устройства Down (Down Device Object).** Это — "боковой" объект, в том смысле, что он не установлен в стеке устройства режима ядра. Объект Down принимает запросы ввода/вывода от хост-процесса UMDF после их обработки драйвером. Он передает эти запросы нижележащему стеку устройства, который осуществляет взаимодействие с устройством. Хост-процесс также взаимодействует с объектом устройства Down для создания интерфейса устройства или имперсонации клиента.

Отражатель также создает еще один "боковой" объект — объект устройства управления (control device object), который обслуживает все хост-процессы UMDF в системе. Объект устройства управления контролирует взаимодействие между отражателем и хост-процессами UMDF, которые не относятся к запросам ввода/вывода, такими как, например, создание или прекращение хост-процесса.

### **Зачем нужны "боковые" объекты?**

На этот вопрос можно попробовать ответить, приведя пару простых примеров. Начиная работу над стеком устройства, нам был нужен драйвер UMDF, для того чтобы иметь возможность посылать запросы ввода/вывода нижней части стека устройства. Мы не могли общаться с нижней частью стека самой, т. к. стек устройства нельзя открыть, не запустив сначала менеджер ввода/вывода.

Подобная проблема происходит и при удалении устройства. Менеджер PnP не будет посыпать стеку устройства запрос на удаление до тех пор, пока не будет закрыт последний дескриптор устройства. Но если этот дескриптор закрыт, драйвер пользовательского режима не сможет посыпать команды устройству во время его удаления.

Мы могли бы создать внутренний интерфейс, который бы позволил нам посыпать запрос ввода/вывода на раннем этапе, но такой подход не работал бы с `ReadFile` и `WriteFile` и создавал бы проблемы совместимости. "Боковой" объект представляет элегантное решение этой проблемы. Так как он не находится в стеке устройства, его можно открыть в любое время. Любой ввод/вывод от процессов, не являющихся хост-процессами, блокируется отражателем, поэтому никто больше не может открыть или использовать "боковой" объект. Драйвер также может использовать для обращения к объекту во время его запуска и удаления те же самые функции API, как и в любое другое время. (*Питер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

**Нижний стек устройства (lower device stack).** В конечном счете, для осуществления фактического взаимодействия с устройством, драйверы UMDF зависят от подлежащего стека устройства режима ядра. В случаях некоторых типов драйверов, например, таких как драйверы USB-устройств, драйверы UMDF могут использовать стека устройства режима ядра, предоставляемого компанией Microsoft. В противном случае, нижнюю часть стека должен предоставить поставщик устройства.

## **Критические ошибки в драйверах UMDF**

Когда инфраструктура останавливает драйвер по причине критической ошибки, останавливается только хост-процесс драйвера, а все остальные процессы и сама система продолжают работать. Большим преимуществом исполнения в пользовательском режиме является то, что драйверу разрешен доступ только к своему собственному адресному пространству, но не к адресному пространству режима ядра. В результате, критическая ошибка драйвера не влияет на память ядра и вызывает сбой только процесса драйвера, а не всей системы.

При сбое в UMDF-драйвере отражатель завершает все ожидающие исполнения запросы ввода/вывода со статусом ошибки и посыпает извещение всем приложениям, зарегистрировавшимся для получения таких извещений. Windows регистрирует сбой и выполняет пять попыток перезапустить экземпляр устройства. Если устройство нельзя перезапустить, система отключает устройство и, если не существует открытых дескрипторов на какие-либо объекты устройств в стеке устройства, выгружает соответствующие драйверы ядра из стека устройства.

Дополнительную информацию по обработке критических ошибок в драйверах UMDF см. в разделе **Handling Driver Failures** (Обработка критических ошибок) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79794>.

## Типичный запрос ввода/вывода UMDF

Приложение начинает взаимодействие с драйвером UMDF с получения имеми символьной ссылки устройства с помощью интерфейса устройства, после чего вызывает функцию `CreateFile` для создания дескриптора устройства. Приложение использует полученный дескриптор, чтобы посыпать драйверу запросы ввода/вывода. Обычно приложение не "знает", что оно использует драйвер пользовательского режима, т. к. процедура для получения дескриптора устройства и отправки запросов ввода/вывода работает точно таким же образом для драйверов UMDF, как и для драйверов режима ядра.

Исполнение типичного запроса ввода/вывода UMDF, такого как, например, запроса на чтение или запроса IOCTL, протекает следующим образом:

1. Приложение вызывает функцию Windows, такую как, например, `ReadFile` или `DeviceIoControl`, чтобы послать устройству запрос ввода/вывода. Windows вызывает соответствующую процедуру ввода/вывода режима ядра, которая передает запрос менеджеру ввода/вывода.
2. Менеджер ввода/вывода создает пакет IRP и посыпает его объекту устройства Up рефлектора, который посыпает запрос хост-процессу драйвера. Хост-процесс создает пакет IRP пользовательского режима для представления пакета IRP в части пользовательского режима стека устройства.
3. Хост-процесс передает пакет IRP пользовательского режима верхнему экземпляру инфраструктуры в стеке устройства.
4. Инфраструктура использует данные с пакета IRP пользовательского режима для создания инфраструктурного объекта запроса ввода/вывода и передает его соответствующему объекту обратного вызова драйвера.
5. Драйвер обрабатывает запрос ввода/вывода и посыпает его стандартному исполнителю ввода/вывода драйвера, которым обычно является следующий нижележащий драйвер в стеке. В результате этого действия запрос пересыпается обратно инфраструктуре, которая запрашивает хост-процесс посыпать запрос следующему драйверу. Инфраструктура также сохраняет указатель на объект запроса, с помощью которого инфраструктура позже определяет, зарегестрировал ли драйвер обратный вызов для завершения запроса.
6. Если в стеке есть другие драйверы UMDF, шаги 3—5 выполняются последовательно для каждого драйвера до тех пор, пока запрос не дойдет до драйвера, который может выполнить запрос.
7. Хост-процесс посыпает обработанный IRP-пакет объекту устройства Down отражателя, который создает второй пакет IRP.

8. Объект устройства Down передает запрос нижнему стеку устройства, который обрабатывает запрос и осуществляет необходимое взаимодействие с устройством. В конечном счете, драйвер в нижнем стеке завершает второй пакет IRP.
9. Объект устройства Down возвращает выполненный запрос ввода/вывода хост-процессу драйвера. Если какие-либо драйверы UMDF зарегистрировали обратный вызов для завершения запроса, инфраструктура вызывает их по порядку, начиная с нижней части стека.
10. После завершения обработки обратных вызовов для всех UMDF-драйверов инфраструктура передает завершенный запрос ввода/вывода объекту устройства Up отражателя и освобождает пакет IRP пользовательского режима. Объект устройства Up передает запрос обратно менеджеру ввода/вывода, который возвращает результаты исполнения запроса приложению.

## Инфраструктура KMDF

Инфраструктура KMDF служит в качестве слоя абстракции над WDM. Поскольку как и инфраструктура KMDF, так и соответствующие драйверы исполняются в режиме ядра, ее архитектура намного проще, чем архитектура инфраструктуры UMDF. Схема инфраструктуры KMDF представлена на рис. 4.3. В качестве примера, на рисунке показано устройство простого стека устройства, который состоит из драйвера функции KMDF, драйвера фильтра WDM и драйвера шины WDM.

## Компоненты инфраструктуры KMDF

Инфраструктура KMDF состоит из собственно инфраструктуры, драйверов KMDF и других драйверов.

**Инфраструктура.** В противоположность нескольким компонентам среды исполнения, составляющим инфраструктуру UMDF, инфраструктура KMDF состоит только из одного компонента среды исполнения — самой инфраструктуры. Один экземпляр инфраструктуры предоставляет поддержку всем драйверам KMDF системы. Если на системе установлены несколько основных версий инфраструктуры, для каждой из них создается экземпляр, чтобы предоставить поддержку драйверам, скомпилированным под данную версию инфраструктуры.

Каждый драйвер KMDF вызывает инфраструктуру для создания инфраструктурного объекта устройства для каждого поддерживаемого им устройства. Инфраструктура, в свою очередь, создает соответствующий объект устройства WDM и устанавливает его в стеке устройств для устройства. Для драйверов функции KMDF инфраструктура создает объект FDO и устанавливает его сверху любых расположенных ниже драйверов фильтра, и т. д. Так как верхняя и нижняя кромки драйверов KMDF работают подобно драйверу WDM, стек устройств может содержать смесь драйверов KMDF и WDM.

Когда принадлежащий инфраструктуре объект получает от менеджера ввода/вывода пакет IRP, для обработки запроса инфраструктура вступает во взаимодействие с соответствующим драйвером KMDF. Если данный драйвер KMDF не может выполнить запрос, инфраструктура возвращает пакет IRP обратно менеджеру ввода/вывода для передачи его следующему лежащему ниже драйверу в стеке, и т. д. Кроме управления потоком пакетов IRP, инфраструктура также поддерживает DDI-интерфейс KMDF и объектную модель, используемую

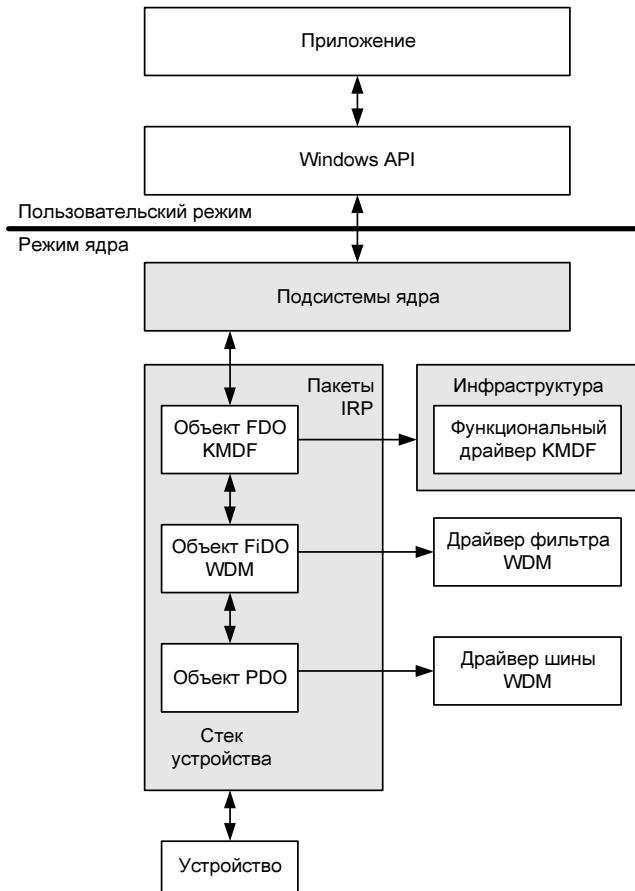


Рис. 4.3. Схема инфраструктуры и драйвера KMDF

драйверами для взаимодействия с инфраструктурой, другими драйверами и устройством. Инфраструктура также отслеживает состояние системы и устройства и предоставляет стандартную обработку запросам Plug and Play и управления энергопотреблением.

**Драйвер KMDF.** Модель KMDF до некоторой степени подобна модели "порт — минипорт", где инфраструктура играет роль драйвера порта, а драйвер KMDF — драйвера минипорта. Объект устройства инфраструктуры устанавливается в стеке устройств, и инфраструктура осуществляет многие из функций драйвера. Но для специфичной для устройства обработки инфраструктура зависит от соответствующего драйвера KMDF. Но между моделью "порт — минипорт" и KMDF есть одна большая разница: KMDF не накладывает ограничений на то, какие процедуры DDI-интерфейса драйвер может вызывать, или на способ работы драйвера.

**Другие драйверы.** Стек устройства может состоять из нескольких драйверов, любой или все из которых могут быть драйвером KMDF. В примере на рис. 4.3 показан драйвер функции KMDF, который установлен сверху драйвера фильтра WDM. Но с таким же успехом это могли быть два драйвера KMDF или драйвер функции WDM сверху драйвера фильтра KMDF. Пакеты IRP перемещаются по стеку обычным образом, и инфраструктура создает

соответствующие инфраструктурные объекты запроса, когда они достигают связанного с KMDF объекта устройства.

## Критические ошибки в драйверах KMDF

Когда инфраструктура останавливает драйвер по причине критической ошибки, целостность системы непоправимо нарушается, поэтому инфраструктура генерирует останов bugcheck и вызывает сбой системы. Все остановы bugcheck, генерируемые инфраструктурой, используют код WDF\_VIOLATION и имеют четыре параметра. В первом параметре указывается конкретный тип ошибки, а в остальных параметрах предоставляется дополнительная информация об ошибке.

Дополнительную информацию по вопросу останова bugcheck см. в разделе **Interpreting Bug Check Codes** (Интерпретирование кодов bug check) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82320>.

## Типичный запрос ввода/вывода KMDF

Приложение начинает взаимодействие с драйвером KMDF точно так же, как и в случае с другими типами драйверов: с получения имени символьной ссылки устройства с помощью интерфейса устройства, после чего вызывает функцию CreateFile для создания дескриптора устройства. Исполнение типичного запроса ввода/вывода протекает следующим образом:

1. Приложение вызывает функцию Windows, такую как, например, ReadFile или WriteFile, чтобы послать устройству запрос ввода/вывода.

Windows вызывает соответствующую процедуру ввода/вывода режима ядра, которая передает запрос менеджеру ввода/вывода.

2. Менеджер ввода/вывода создает пакет IRP и передает его наверх стека устройства.

В данном примере наверху находится объект FDO, связанный с драйвером KMDF, поэтому он и получает запрос.

3. Если объект FDO может обработать данный тип пакета IRP, инфраструктура преобразовывает его в инфраструктурный объект запроса, который она передает драйверу KMDF.

4. Драйвер обрабатывает запрос и возвращает результаты инфраструктуре.

В случае успешного или неуспешного завершения запроса, инфраструктура завершает пакет IRP и возвращает результаты менеджеру ввода/вывода. В противном случае инфраструктура форматирует должным образом пакет IRP и передает его следующему нижележащему драйверу для дальнейшей обработки. Если драйвер KMDF должен выполнить какую-либо заключительную обработку после завершения запроса, он регистрирует в инфраструктуре обратный вызов завершения ввода/вывода, который, в свою очередь, устанавливает в пакете IRP процедуру завершения ввода/вывода.

5. Если драйвер KMDF передает запрос вниз по стеку для дальнейшей обработки, инфраструктура форматирует пакет IRP должным образом и возвращает его менеджеру ввода/вывода.

Менеджер ввода/вывода передает пакет IRP вниз драйверу фильтра WDM, который обрабатывает запрос, и так далее.

6. Когда запрос, наконец, выполнен, если драйвер KMDF зарегистрировал обратный вызов завершения ввода/вывода, менеджер ввода/вывода вызывает процедуру завершения,

установленную инфраструктурой. Инфраструктура, в свою очередь, вызывает обратный вызов завершения ввода/вывода драйвера, с тем, чтобы драйвер мог завершить свою обработку.

## Поддержка устройств и драйверов в WDF

Какую из этих двух инфраструктур — UMDF или KMDF — использовать для устройства, зависит в первую очередь от типа устройства. В этом разделе в общих чертах обсуждаются типы устройств, поддерживаемые инфраструктурами, и вопросы, которые необходимо учесть при решении в пользу той или другой инфраструктуры.

Полный список устройств, поддерживаемых UMDF и KMDF, можно просмотреть в разделе **FAQ: Questions from Driver Developers about Windows Driver Foundation** (FAQ: вопросы, задаваемые разработчиками драйверов о Windows Driver Foundation) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=81576>.

### Устройства, поддерживаемые UMDF

Инфраструктура UMDF поддерживает разработку драйверов для протокольных устройств или устройств на основе последовательной шины, таких как, например, устройства USB. Инфраструктуру UMDF можно использовать для создания драйверов для 32- и 64-битных устройств для x86- и x64-версий Windows соответственно. Примеры некоторых таких устройств приведены в следующем списке:

- ◆ портативные устройства, например, карманные компьютеры (PDA), сотовые телефоны и плееры;
- ◆ устройства USB (за исключением изохронных устройств);
- ◆ вспомогательные видеоустройства.

Устройство может быть подключено напрямую, по сети или с помощью радиосвязи, например, Bluetooth. UMDF можно также использовать для создания драйверов для чисто программных устройств.

#### 32- и 64-битные драйверы UMDF

Нас часто спрашивают, можно ли разрабатывать 32-битный драйвер UMDF на 64-битной системе. С драйверами WDM и KMDF это нельзя, т. к. 32-битный код не может исполняться в 64-битном ядре. Но так как 32-битные пользовательские приложения могут исполняться на 64-битных системах, по аналогии возникает вопрос о такой возможности и для драйверов UMDF.

Это невозможно. Хотя 32-битный код драйвера и мог бы исполняться в UMDF, он не может работать с запросами от 64-битных приложений. Точно так же, как для 64-битных драйверов часто необходим специальный код для обработки запросов IOCTL от 32-битных приложений, для 32-битного драйвера на 64-битной системе обработку запросов от 64-битных приложений было бы необходимо выполнять на специальной основе. UMDF не может пропустить такие запросы просто так, т. к. это могло бы вызвать переполнение буфера или другую ошибку безопасности.

По-моему, отсутствие такой возможности — не такая уж и большая потеря. Поскольку драйвер и так нужно переписывать для исполнения в пользовательском режиме, то почему бы в то же самое время не скомпилировать его под 64-битный режим? Кроме этого, в поведении 32- и 64-битных приложений имеются некоторые тонкие различия, и поэтому вам все

равно следовало бы протестировать его на 64-битной системе. При таком раскладе, почему бы просто не разрабатывать 64-битный драйвер с самого начала? (Питер Виленд (*Peter Wieland*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Достоинства драйверов UMDF

Драйверы UMDF имеют многочисленные преимущества над драйверами режима ядра.

**Среда исполнения драйвера.** Среда исполнения драйверов UMDF намного проще, чем драйверов режима ядра. Например, при написании кода драйверов режима ядра необходимо принимать меры во избежание проблем, связанных с IRQL, ошибками страницы и контекстом потока. В пользовательском же режиме эти проблемы отсутствуют. Драйверы UMDF всегда могут справиться с ошибками страницы. Кроме этого, драйверы UMDF всегда исполняются в адресном пространстве, отдельном от запрашивающего процесса.

**Повышенная стабильность системы.** В отличие от драйверов режима ядра, среда исполнения драйверов UMDF изолирована от остальной системы. В частности, каждый драйвер UMDF исполняется в собственном адресном пространстве, целостность которого не может быть нарушена, например, попыткой другого драйвера записать в нее свои данные. Хотя драйвер UMDF, содержащий ошибки или целостность которого была нарушена, может сделать свое устройство неработоспособным, возможность того, что драйвер UMDF может вызвать проблемы системного масштаба, намного ниже, чем для драйверов режима ядра. В частности, драйверы UMDF не могут создать условий для останова bugcheck, т. к. они не могут обращаться к адресному пространству ядра и не могут вызывать функции режима ядра, которые манипулируют важными структурами данных системы.

**Пониженная угроза безопасности.** Драйверы UMDF обычно исполняются с учетной записью LocalService, обладающей только минимальными привилегиями. Это обстоятельство делает драйверы UMDF относительно непривлекательной целью для атак. Даже формально успешная атака драйвера UMDF будет не слишком удачной в смысле получения доступа к системным ресурсам, т. к. сам драйвер имеет очень ограниченный доступ к ним. Подвержены опасности будут лишь данные, проходящие через захваченный драйвер. Кроме этого, вероятность того, что драйвер может подвесить или вызвать сбой системы, таким образом создавая DoS-ситуацию, намного ниже для драйверов UMDF. В самом худшем случае, только целостность самого процесса драйвера может быть нарушена, но его можно остановить, не влияя на другие процессы.

### Внимание!

Драйверы UMDF могут поддерживать имперсонацию. Однако драйвер должен зарегистрироваться для использования этой возможности при его установке, а также получить разрешение от запрашивающего приложения имперсонировать пользователя. Хотя эти обстоятельства и уменьшают фактор риска, тем не менее с драйверами UMDF, поддерживающими имперсонацию, все равно необходимо быть осторожным. Например, безопасность драйвера можно нарушить, инжектировав в него код, рассчитанный на использование имперсонации. Когда такой драйвер получит запрос от администратора, разрешающий имперсонацию, этот код имперсонирует администратора и атакует систему под его именем.

**Windows API.** Большинство разработчиков приложений знакомо с программным интерфейсом Windows API. Хотя драйверы UMDF не могут вызывать функции ядра, они способны вызывать большую часть функций программного интерфейса Windows API, который предоставляет доступ к сервисам, недоступным в режиме ядра, например шифрование. Но про-

цессы драйверов UMDF исполняются в сессии, не позволяющей взаимодействия пользователя, поэтому они не могут использовать те функции Windows API, которые поддерживают пользовательские интерфейсы.

Так как Windows API является компонентом пользовательского режима, прежде чем выполнять изменения, запрашиваемые из пользовательского режима, Windows выполняет дополнительные проверки безопасности и верификации. Тем не менее при вызове внешних процессов или компонентов необходимо соблюдать осторожность. Например, загрузка в драйвер скомпрометированного объекта COM в виде внутрипроцессного компонента создаст брешь в безопасности системы. Также, Windows API может быть не очень подходящей для применения с асинхронной моделью ввода/вывода. Например, долго исполняющаяся синхронная операция, которую нельзя аннулировать, может заставить драйвер обрабатывать свои запросы синхронно, таким образом понижая производительность драйвера.

### **Интерфейсы пользователя и драйвера**

Не пытайтесь снабдить ваш драйвер возможностью выводить пользовательский интерфейс. Только потому, что драйвер исполняется в пользовательском режиме, не означает, что с его помощью можно выводить элементы пользовательского интерфейса. Драйверы UMDF исполняются в нулевой сессии (session 0), вместе с другими системными сервисами, и не имеют доступа к рабочему столу. Это означает не только то, что пользователь не может видеть никаких элементов пользовательского интерфейса, которыми вы можете снабдить свой драйвер, но и то, что если какой-либо из этих элементов заблокируется, ожидая ввод от пользователя, ваш драйвер зависнет, т. к. пользователь не может этого сделать.

Если вы хотите снабдить свой драйвер пользовательским интерфейсом, делайте это с помощью отдельного компонента интерфейса, который взаимодействует с драйвером посредством вызовов ввода/вывода Windows. (*Петер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

**Отладка в пользовательском режиме.** Для отладки драйверов UMDF можно пользоваться отладчиком пользовательского режима вместо отладчика режима ядра. Разработка и отладка драйвера в пользовательском режиме может занять меньше времени, т. к. ошибки затрагивают только текущий процесс, а не всю систему. Таким образом, экономится время на перезагрузке системы после сбоя, вызванного ошибкой в драйвере режима ядра. Кроме этого, для отладки драйверов UMDF требуется всего лишь один компьютер, а не два, как обычно необходимо для драйверов режима ядра.

**Программирование на C++ или C.** Так как драйверы UMDF основаны на технологии COM, их без проблем можно писать на языке C++. Можно также использовать для этой цели язык C, но это не очень распространенная практика.

**Производительность на уровне драйверов режима ядра.** Для типов устройств, которые драйверы UMDF могут поддерживать, фактором, ограничивающим скорость, обычно является пропускная способность ввода/вывода, а не внутренняя производительность драйвера. Для таких устройств производительность драйверов UMDF сравнима с эквивалентными драйверами KMDF.

### **Недостатки драйверов UMDF**

Хотя исполнение в пользовательском режиме предоставляет многие преимущества, ограничение на доступ к адресному пространству режима ядра имеет определенные недостатки. Некоторые типы драйверов просто не могут исполняться в пользовательском режиме, и их

нельзя реализовать в виде драйверов UMDF. Если драйвер требует следующие возможности и ресурсы, его необходимо реализовывать как драйвер режима ядра:

- ◆ прямой доступ к аппаратной части, включая способность обрабатывать прерывания;
- ◆ бесперебойные временные циклы;
- ◆ доступ к структурам данных или памяти ядра.

Кроме этого, драйвер UMDF не может быть клиентом ядра Windows или драйвера режима ядра.

## Устройства, поддерживаемые KMDF

Инфраструктура KMDF разрабатывалась как замена модели WDM, поэтому ее можно применять для разработки драйверов для таких устройств и классов устройств, как WDM. Исключениями являются драйверы для классов устройств, поддерживаемых некоторыми моделями мини-портов, например, драйверы Storport.

Как правило, драйвер, отвечающий требованиям WDM, предоставляющий точки входа для основных рабочих процедур ввода/вывода и обрабатывающий пакеты IRP, можно также реализовать, как драйвер KMDF. Для некоторых типов устройств драйверы класса устройств и драйверы портов предоставляют рабочие процедуры драйвера и осуществляют обратный вызов драйвера мини-порта (который, в сущности, представляет собой специфичную для устройства библиотеку функций обратного вызова) для обработки специфических деталей ввода/вывода. Версия инфраструктуры, входящая в состав набора WDK версии Windows Server Longhorn, не поддерживает драйверы и типы устройств с архитектурой Windows Image Acquisition (WIA).

Вопрос переноса существующих драйверов WDM в KMDF обсуждается в разделе **WDM to KMDF Porting Guide** (Руководство по переносу драйверов WDM в KMDF) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=82319>.

### KMDF и WDM

Что делает KMDF общим решением, способным работать с различными стеками устройств, так это то обстоятельство, что инфраструктура выглядит как объект устройства WDM с интерфейсами WDM вверху и внизу. В этом и заключается настоящая мощь KMDF в смысле совместимости и применимости для многих классов устройств.

Но с другой стороны, эта возможность также является одной из наибольших обуз для разработчиков и тестировщиков KMDF. Абстракции и объекты, реализуемые инфраструктурой, должны быть совместимы с многочисленными классами устройств, которые уже выпущены и которые нельзя изменить, чтобы они соответствовали измененной инфраструктуре. Громадный объем работы был проделан по проектированию и тестированию внешних интерфейсов, например, обратных вызовов, связанных с энергопотреблением, чтобы обеспечить всеохватывающую совместимость. (*Даун Холэн (Down Holan), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Как выбрать правильную инфраструктуру

Хотя выбор, какую инфраструктуру использовать, может показаться как принятие очень важного решения, сделать его в действительности очень легко. Если для вашего устройства будет достаточно драйвера UMDF, его и разрабатывайте. Это позволит вам сэкономить время на разработку и отладку, и драйвер будет более стабильным и безопасным. Создавай-

те драйверы режима ядра только тогда, когда в этом есть необходимость. Для большинства типов устройств решить, какой тип драйвера использовать, очень просто. Например:

- ◆ для MP3-плеера подойдет только драйвер UMDF, т. к. в нем применяется модель Windows Portable Devices;
- ◆ типичный сетевой адаптер обрабатывает прерывания и обращается напрямую к памяти, поэтому для него необходим драйвер KMDF.

Несколько типов устройств, наиболее важными из которых являются устройства USB, поддерживаются обеими инфраструктурами. Для большинства устройств USB будет достаточно драйвера UMDF. Но если драйверу требуются сервисы, возможные только с KMDF, тогда необходимо создавать драйвер KMDF.

### Примечание

Ни KMDF, ни UMDF не поддерживают разработку драйверов файловой системы и драйверов фильтра файловой системы. Для разработки драйверов файловой системы необходим устанавливаемый набор для файловых систем, поставляемый вместе с WDK.

# ГЛАВА 5

## Объектная модель WDF

WDF определяет формальную объектную модель, в которой объекты представляют распространенные абстракции компонентов драйверов, например, устройство, запрос ввода/вывода или очередь. Драйвер взаимодействует с этими объектами, а не с базовыми элементами операционной системы Windows. Некоторые объекты создаются инфраструктурой в ответ на внешние события, например прибытие запроса ввода/вывода, а другие объекты создаются самим драйвером.

Хотя эти объекты реализуются по-разному в каждой инфраструктуре, обе инфраструктуры, как KMDF, так и UMDF, разделяют одинаковую объектную модель. Эта объектная модель и ее реализация и рассматриваются в данной главе.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b> Comsup.h	%wdk%\src\umdf\usb\fx2_driver\final
<b>Образцы драйверов</b> Echo Fx2_Driver Osrusbf2 WpdHelloWorldDriver	%wdk%\src\kmdf\Echo\sys %wdk%\src\umdf\usb\fx2_driver\final %wdk%\src\kmdf\osrusbf2\sys\final %wdk%\src\umdf\wpd\WpdHelloWorldDriver
<b>Документация WDK</b> Объекты и интерфейсы UMDF Объекты драйвера режима ядра инфраструктуры	<a href="http://go.microsoft.com/fwlink/?LinkId=79583">http://go.microsoft.com/fwlink/?LinkId=79583</a> <a href="http://go.microsoft.com/fwlink/?LinkId=79584">http://go.microsoft.com/fwlink/?LinkId=79584</a>

## Обзор объектной модели

Объектная модель WDF определяет набор объектов, которые представляют распространенные абстракции компонентов драйверов, например, устройство, запрос ввода/вывода или очередь. Независимо от типа драйверной модели, почти каждый драйвер работает с этими абстракциями или в виде структур данных, или в виде внутренних шаблонов, создаваемых самим драйвером. Например, в драйвере WDM структура данных пакета IRP представляет

операцию ввода/вывода. Драйвер манипулирует этой структурой, комбинируя прямой доступ и вызовы интерфейса DDI менеджера ввода/вывода Windows. Подобным образом драйверы WDM также создают внутренние очереди для управления потоком запросов ввода/вывода.

Модель WDF отличается от модели WDM и других драйверных моделей тем, что в ней определяются формальные объекты для представления этих структур данных и шаблонов. Объект запроса ввода/вывода модели WDF представляет запрос ввода/вывода, а объект очереди ввода/вывода модели WDF представляет очередь. Драйверы WDF взаимодействуют с этими объектами только через интерфейс DDI инфраструктуры.

Все объекты имеют базовый набор свойств и подчиняются систематическим правилам для управления временем жизни, управления контекстом, шаблонов обратного вызова и иерархии объектов. В объектной модели WDF:

- ◆ объекты играют роль стандартных блоков, непрозрачных для драйвера. Драйвер модифицирует эти объекты посредством четко определенных интерфейсов;
- ◆ объекты имеют методы, свойства и события. Стандартная реакция для каждого события определяется инфраструктурой. Для предоставления поддержки специфичного для устройства поведения драйвер содержит функции обратного вызова для событий, которые заменяют стандартные функции обработки событий;
- ◆ объекты организованы в иерархию и имеют четко определенные жизненные циклы. Когда удаляется родитель объекта, объект также удаляется;
- ◆ любой объект может иметь область контекста, в которой драйвер сохраняет информацию, необходимую ему для обслуживания объекта.

Формат и содержимое этой области контекста полностью определяются драйвером.

## Методы, свойства и события

Драйверы взаимодействуют с объектами WDF посредством методов (функций), свойств (данных) и специфичных для объекта событий, для обработки которых драйвер может предоставлять функции обратного вызова.

- ◆ *Методы* выполняют какие-либо действия над объектом.
- ◆ *Свойства* описывают характеристики объекта. С каждым свойством связан метод для получения значения свойства и, если применимо, отдельный метод для установки его значения. Считывание и установка значения некоторых свойств всегда выполняется успешно, но для некоторых свойств эти операции могут быть неудачными. Функции, чьи имена содержат Set и Get, всегда выполняются успешно. А функции, чьи имена содержат Assign и Retrieve, могут завершиться неудачно, возвращая код статуса.
- ◆ *События* являются извещениями, указывающими, что произошло что-то, возможно, требующее вмешательства со стороны драйвера. Драйверы могут регистрировать обратные вызовы для определенных событий, представляющих для них важность. Инфраструктура активирует эти обратные вызовы для событий драйвера и обрабатывает все другие события стандартным образом.

### Именование методов

Мы много думали о том, каким образом именовать методы, и в случае со свойствами нам пришлось порядочно поломать голову над этим вопросом. Некоторое время мы рассматривали

вали вариант, при котором функции свойств должны были бы проверять достоверность дескрипторов объектов и возвращать ошибку в случае передачи им недействительного объекта. Но это влечет за собой или непроверенные обработчики ошибок или проигнорированные коды ошибок.

В конце концов, нам пришлось разделить свойства на две группы — свойства, которые всегда можно прочитать и установить, и те, с которыми это невозможно. Мы решили назвать функции для этих двух типов свойств по-разному, с тем, чтобы человек, изучающий инфраструктуры, мог бы с легкостью запомнить, что они делают, и чтобы ему не требовалось проверять возвращаемый код. Функции `Get` и `Set` всегда выполняются успешно (при условии действительных данных ввода). А функции `Assign` и `Retrieve` требуют проверки возвращаемого кода. (*Питер Вилант (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Обратные вызовы по событию

Обратные вызовы по событию предоставляют специфичные для устройства и для драйвера ответы на события. Каждый обратный вызов по событию создается для конкретного типа объекта. Например, объекты устройств поддерживают события Plug and Play, и драйвер может "зарегистрироваться" для этих событий при создании объекта устройства. Некоторые события, такие как, например, добавление нового устройства, требуют специфичный для драйвера обратный вызов.

Обратные вызовы по событию инициируются одинаково как в UMDF, так и KMDF, но драйверы для каждой инфраструктуры реализуют их по-разному:

- ◆ Драйвер UMDF создает и сопоставляет объект обратного вызова с каждым объектом инфраструктуры. Этот объект обратного вызова создает один или несколько интерфейсов обратных вызовов. Инфраструктура определяет возможные интерфейсы обратных вызовов и с помощью COM-метода `QueryInterface` выясняет, какие интерфейсы драйвер реализует для определенного объекта, инициирующего обратный вызов.
- ◆ Драйвер KMDF поддерживает обратный вызов по событию, реализуя функцию обратного вызова и предоставляя указатель на эту функцию при инициализации соответствующего объекта, обычно в специфичной для объекта структуре конфигурации.

Таким образом, обратные вызовы по событию драйверов KMDF представляют собой элементы в таблице функций, в то время как функции обратного вызова по событию драйверов UMDF являются методами интерфейса, реализованного в объекте обратного вызова.

Когда происходит событие, инфраструктура активирует обратный вызов. Например, неожиданное удаление устройства является событием Plug and Play. Если устройство может быть удалено без предупреждения, его драйвер может реализовать обратный вызов для события неожиданного удаления для выполнения необходимых операций. Когда PnP-менеджер посыпает извещение о неожиданном удалении устройства, инфраструктура активирует обратный вызов по событию неожиданного удаления, принадлежащий драйверу данного устройства.

## Атрибуты объектов

Объектная модель WDF задает набор общеупотребляемых атрибутов объектов. При создании объекта драйвер устанавливает значения атрибутов (или принимает значения по умолчанию, предоставляемые инфраструктурой).

Следующие атрибуты могут применяться как к UMDF-, так и к KMDF-объектам:

- ◆ объект, являющийся родителем объекта;
- ◆ область синхронизации для обратных вызовов для определенных событий ввода/вывода;
- ◆ область контекста объекта;
- ◆ обратный вызов для освобождения ссылок перед удалением объекта.

Драйверы UMDF указывают значения атрибутов в индивидуальных параметрах в соответствующих методах создания объектов.

Драйверы KMDF предоставляют значения атрибутов в отдельно инициализированной структуре, которая является параметром для всех методов создания объектов. С некоторыми типами объектов KMDF могут применяться следующие дополнительные атрибуты:

- ◆ максимальный уровень IRQL, при котором инфраструктура инициирует определенные обратные вызовы по событию;
- ◆ информация об области контекста, включая ее размер и тип;
- ◆ обратный вызов для освобождения ресурсов при уничтожении объекта.

## Иерархия и время жизни объектов

Объекты WDF организованы в иерархию. За исключением объекта драйвера WDF, каждый объект имеет родителя — следующего вышележащего объекта в иерархии. Корневым объектом иерархии является объект драйвера WDF; все другие объекты иерархии являются его подчиненными. Для некоторых типов объектов, при создании объекта драйвер может указать его родителя. Но объекты других типов имеют предопределенных родителей, изменить которых нельзя.

Объектная модель WDF была разработана с целью позволить драйверам полагаться на инфраструктуры в управлении временем жизни объекта, зачистке объектов и состоянием объектов, при минимальном участии самого драйвера. Устанавливая соответствующие взаимоотношения между родителями и потомками, драйвер обычно может избежать использования прямых ссылок на объекты или удаления объектов явным образом. Вместо этого, для удаления объектов в требуемое время он может полагаться на объектную иерархию.

## Контекст объекта

Ключевой особенностью объектной модели WDF является область контекста объекта. Область контекста объекта представляет собой определенную драйвером область памяти для данных, которые драйвер использует с определенным объектом:

- ◆ драйверы UMDF обычно размещают данные контекста в объекте обратного вызова;
- ◆ драйверы KMDF могут создавать область контекста, являющуюся частью инфраструктурного объекта.

Размер и структура области контекста объекта определяется драйвером. Например, драйвер может создать область контекста для запроса ввода/вывода, полученного от инфраструктуры, и сохранить назначенное драйвером событие, используемое с запросом, в области контекста.

## Реализация объектной модели UMDF

Инфраструктура UMDF содержит набор инфраструктурных объектов, а драйвер создает объект обратного вызова, соответствующий каждому инфраструктурному объекту, для которого драйверу требуется обратный вызов по событию. Созданные драйвером объекты обратного вызова предоставляют интерфейсы обратных вызовов, определенных инфраструктурой.

Драйверы UMDF используют следующие типы объектов:

- ◆ созданный драйвером файл;
- ◆ параметры завершения ввода/вывода;
- ◆ базовый объект;
- ◆ получатель ввода/вывода;
- ◆ драйвер;
- ◆ память;
- ◆ устройство;
- ◆ устройство USB;
- ◆ файл;
- ◆ получатель ввода/вывода USB;
- ◆ запрос ввода/вывода;
- ◆ интерфейс USB;
- ◆ очередь ввода/вывода.

Объекты и интерфейсы UMDF основаны на программном шаблоне COM. Изо всей инфраструктуры и библиотеки времени исполнения COM в UMDF используется только интерфейс запросов и счетчик ссылок.

Информация о COM приводится в *главе 18*.

## Соглашение об именах UMDF

В UMDF применяется соглашение об именах реализуемых ею интерфейсов и определяемых в ней интерфейсов обратных вызовов.

Реализуемые в UMDF интерфейсы именуются в следующем формате:

IWDFObject[ Описание ]

Здесь:

- ◆ *Объект* указывает объект UMDF, предоставляющий интерфейс;
- ◆ *Описание* предоставляет информацию о том, на какой аспект объекта распространяется действие интерфейса.

Например, инфраструктурные объекты устройства предоставляют интерфейс IWDFDevice, который поддерживает методы манипулирования инфраструктурным объектом устройства. А инфраструктурные объекты получателя ввода/вывода предоставляют интерфейсы IWDFIoTarget и IWDFIoTargetStateManagement.

Методы интерфейсов UMDF именуются в следующем формате:

*Действие**Спецификатор*

Здесь:

- ◆ *Действие* указывает действие, выполняемое функцией;
- ◆ *Спецификатор* указывает объект или данные, на которые направлено действие метода.

Например, интерфейс IWDFDevice содержит методы CreateIoQueue и ConfigureRequestDispatching.

Свойства интерфейсов UMDF именуются в следующем формате:

{Set|Get}*Данные*  
{Assign|Retrieve}*Данные*

Здесь *Данные* указывают поле для чтения или записи функцией.

Считывание и установка значения некоторых свойств всегда выполняется успешно, но для некоторых свойств эти операции могут быть неудачными. Функции, чьи имена содержат Set и Get, всегдачитывают и устанавливают свойства успешно. А функции, чьи имена содержат Assign и Retrieve, могут завершиться неудачно, возвращая переменную HRESULT, содержащую код статуса исполнения.

Например, функции IWDFDevice::SetPnpState и IWDFDevice::GetPnpState устанавливают и возвращают значение состояния Plug and Play объекта устройства.

Интерфейсы обратных вызовов по событию объектов UMDF именуются в следующем формате

I*Объект*Callback*Описание*

Здесь:

- ◆ *Объект* указывает тип объекта;
- ◆ *Описание* указывает назначение интерфейса.

Например, интерфейс IQueueCallbackWrite создается для объекта обратного вызова очереди. Он поддерживает метод, вызываемый инфраструктурой в ответ на запрос на запись.

Интерфейсы с именами IPnpCallbackXxx являются исключением. Хотя они содержат PnP в своих именах, они реализуются на объектах устройств. Они поддерживают методы, которые отвечают на события Plug and Play.

Методы UMDF интерфейсов обратного вызова именуются в следующем формате:

On*Событие*

Здесь *Событие* указывает событие, для которого инфраструктура активирует метод обратного вызова.

Например, когда для объекта устройства прибывает запрос на запись, инфраструктура вызывает метод IQueueCallbackWrite::OnWrite. А метод IPnpCallback::OnD0Entry вызывается, когда устройство переходит в рабочее состояние энергопотребления.

## Объекты и интерфейсы инфраструктуры UMDF

Интерфейсы, реализуемые инфраструктурой для каждого типа объектов, приведены в табл. 5.1.

Таблица 5.1. Объекты и интерфейсы инфраструктуры UMDF

Тип объекта	Назначение объекта	Интерфейс	Назначение интерфейса
Базовый объект	Представляет общий базовый объект для использования драйвером необходимым образом	IWDFObject	Поддерживает общие для всех объектов действия, например, блокировку, удаление и управление областью контекста
Устройство	Представляет объект устройства. Драйвер обычно имеет один объект устройства для каждого управляемого им устройства. Потомок объекта IWDFObject	IWDFFDevice	Предоставляет специфичную для устройства информацию и создает специфичные для устройства объекты
		IWDFFFileHandleTargetFactory	Создает объект — получатель ввода/вывода по дескриптору файла
		IWDFFUsbTargetFactory	Создает объект устройства — получателя ввода/вывода USB
Драйвер	Представляет сам драйвер. Каждый драйвер имеет один объект драйвера. Потомок объекта IWDFObject	IWDFDriver	Создает потомков объекта драйвера и предоставляет информацию о версии
Файл	Представляет инфраструктурный объект файла, открытый с помощью функции CreateFile, посредством которого приложения могут обращаться к устройству. Потомок объекта IWDFObject	IWDFFFile	Возвращает информацию о файле и устройстве, сопоставленных с дескриптором файла
Созданный драйвером файл	Представляет инфраструктурный объект файла, созданного драйвером. Потомок объекта IWDFFile	WDFDriverCreatedFile	Закрывает созданный драйвером файл
Очередь ввода/вывода	Представляет очередь ввода/вывода, управляющую потоком ввода/вывода в драйвере. Драйвер может иметь неограниченное количество очередей ввода/вывода. Потомок объекта IWDFObject	IWDFIoQueue	Конфигурирует, управляет и извлекает запросы и информацию из очереди
Запрос ввода/вывода	Представляет запрос на ввод/вывод устройства. Потомок объекта IWDFObject	IWDFIoRequest	Форматирует, посылает, аннулирует и возвращает информацию о запросе ввода/вывода
Информации завершения запроса ввода/вывода	Представляет информацию о завершении для запроса ввода/вывода	IWDFRequestCompletionParams	Возвращает статус, количество байтов и тип запроса для завершенного запроса ввода/вывода
Параметры завершения ввода/вывода	Предоставляет параметры, возвращенные в завершенном запросе ввода/вывода. Потомок объекта IWDFRequestCompletionParams	IWDFIoRequestCompletionParams	Возвращает буферы для запросов ввода/вывода на чтение, запись и управления IOCTL

Таблица 5.1 (окончание)

Тип объекта	Назначение объекта	Интерфейс	Назначение интерфейса
Получатель ввода/вывода	Представляет следующий нежелющий драйвер в стеке устройства, которому драйвер посыпает запросы ввода/вывода. Потомок объекта <code>IWDFObject</code>	<code>IWDFIoTarget</code>	Форматирует и аннулирует запросы ввода/вывода и возвращает информацию о целевом файле
		<code>IWDFIoTargetStateManagement</code>	Выполняет мониторинг и управляет состоянием получателя ввода/вывода
Память	Представляет память, используемую драйвером, обычно в виде буфера ввода или вывода, сопоставленного с запросом ввода/вывода. Потомок объекта <code>IWDFObject</code>	<code>IWDFMemory</code>	Копирует данные в буфер памяти и из буфера и возвращает информацию о буфере
Хранилище свойств	Представляет объект, посредством которого драйвер может сохранять данные в реестре после выгрузки драйвера для использования при последующей его загрузке	<code>IWDFNamedPropertyStore</code>	Позволяет драйверу считывать и записывать информацию в реестр
Интерфейс USB	Представляет интерфейс на устройстве USB. Потомок объекта <code>IWDFObject</code>	<code>IWDFUsbInterface</code>	Возвращает информацию об установках интерфейса USB и выбирает необходимые установки
Устройство получателя USB	Представляет объект устройства USB, являющегося получателем ввода/вывода. Потомок объекта <code>IWDFIoTarget</code>	<code>IWDFUsbTargetDevice</code>	Форматирует запросы для устройства получателя USB и возвращает информацию о нем
Канал получателя USB	Представляет канал USB, являющийся получателем ввода/вывода. Потомок объекта <code>IWDFIoTarget</code>	<code>IWDFUsbTargetPipe</code>	Управляет каналом USB и возвращает информацию о нем
Параметр завершения запроса ввода/вывода USB	Представляет параметры, возвращаемые в завершенном запросе ввода/вывода к получателю USB. Потомок объекта <code>IWDFRequestCompletionParams</code>	<code>IWDFUsbRequestCompletionParams</code>	Возвращает тип запроса и буфера для запросов ввода/вывода на чтение, запись и управления IOCTL для ввода/вывода USB

## Объекты обратного вызова и интерфейсы драйвера UMDF

Драйвер UMDF назначает объект обратного вызова каждому объекту инфраструктуры, чьи события драйвер хочет обрабатывать, и реализует интерфейс обратных вызовов для этих событий на объекте обратного вызова. Каждый драйвер UMDF должен реализовать один объект обратного вызова драйвера для самого драйвера и один объект обратного вызова

устройства для каждого поддерживаемого устройства. Большинство драйверов также реализуют объект обратного вызова очереди для каждой очереди.

Драйвер также должен реализовать интерфейс `IUnknown` для каждого объекта обратного вызова. С этой задачей драйвер может справиться несколькими способами. Образцы драйверов UMDF Echo, Skeleton и USB содержат заголовочный файл `Comsup.h`, в котором интерфейс `IUnknown` реализован в базовом классе `CUnknown`. В вашем собственном драйвере вы можете подключить заголовочный файл `Comsup.h` и указать `CUnknown` в качестве базового класса при объявлении классов драйвера.

Подробная информация о том, как использовать образец `Skeleton` в качестве шаблона для создания своего собственного драйвера, приводится в *главе 13*.

В табл. 5.2 перечислены возможные интерфейсы обратных вызовов, которые могут быть реализованы драйвером.

**Таблица 5.2. Объекты обратного вызова и интерфейсы драйвера UMDF**

Тип объекта	Интерфейс обратного вызова	Назначение интерфейса
Все объекты	<code>IObjectCleanup</code>	Предоставляет обработку, необходимую перед уничтожением объекта, обычно освобождение любых имеющихся циклических ссылок
Драйвер	<code>IDriverEntry</code>	Предоставляет основную точку входа из инфраструктуры в драйвер и методы для инициализации драйвера и добавления его устройств. Это обязательный объект для всех драйверов.  В отличие от других интерфейсов обратных вызовов, драйвер не регистрирует интерфейс <code>IDriverEntry</code> ни с какими конкретными объектами. Вместо этого, инфраструктура запрашивает этот интерфейс у объекта <code>CoClass</code> драйвера, когда она загружает драйвер
Устройство	<code>IPnpCallback</code>	Обрабатывает остановку, удаление и изменения в энергопотреблении устройства
	<code>IPnpCallbackHardware</code>	Предоставляет операции с аппаратурой перед включением и после отключения питания устройства
	<code>IPnpCallbackSelfManagedIo</code>	Позволяет работу драйвера при запуске инфраструктуры и останавливает обработку ввода/вывода при переходах в определенные состояния Plug and Play и управления энергопотреблением
Файл	<code>IFileCallbackCleanup</code>	Обрабатывает запросы на очистку для объектов файла на определенном устройстве. Обычно это требуется, чтобы позволить драйверу отменить любые ожидающие исполнения запросы ввода/вывода для файла, прежде чем закрыть файл
	<code>IFileCallbackClose</code>	Получает извещения о закрытии для объектов файла на определенном устройстве, чтобы драйвер мог освободить выделенные файлу ресурсы

Таблица 5.2 (окончание)

Тип объекта	Интерфейс обратного вызова	Назначение интерфейса
Очередь ввода/вывода	IQueueCallbackCreate	Обрабатывает запросы на создание файла
	IQueueCallbackDefaultIoHandler	Обрабатывает запросы на создание, чтения, записи и IOCTL устройства, для которых не было создано никаких других интерфейсов
	IQueueCallbackDeviceIoControl	Обрабатывает запросы IOCTL устройства
	IQueueCallbackIoResume	Возобновляет обработку запроса ввода/вывода после остановки его очереди
	IQueueCallbackIoStop	Останавливает обработку запроса ввода/вывода при остановке его очереди
	IQueueCallbackStateChange	Извещает драйвер об изменениях в состоянии очереди
	IQueueCallbackRead	Обрабатывает запросы на чтение
Запрос ввода/вывода	IImpersonateCallback	Выполняет операции, связанные с имперсонацией драйвером клиента, издавшего запрос ввода/вывода
	IRquestCallbackCancel	Выполняет операции после отмены запроса ввода/вывода
	IRquestCallbackRequestCompletion	Выполняет операции при завершении запроса ввода/вывода

Когда происходит событие, инфраструктура вызывает методы соответствующего интерфейса обратных вызовов, чтобы драйвер мог ответить на события, сопоставленные с запросом. Например, если драйвер сконфигурировал очередь для запросов на чтение, при получении такого запроса инфраструктура вызывает методы интерфейса `IQueueCallbackRead` объекта обратного вызова очереди.

### Совет

Инфраструктурные объекты и объекты обратного вызова не обязательно должны быть со-поставлены один к одному. Драйвер может делать все, что кажется наиболее разумным с точки зрения обратного вызова и драйвера.

Возьмем, например, обратные вызовы для отмены и завершения ввода/вывода. Эти интерфейсы можно реализовать на любом, кажущемся наиболее подходящим, объекте обратного вызова. Если драйвер имеет очередь, которая также управляет непрерывными операциями чтения на канале прерываний, тогда может быть разумным реализовать интерфейсы обратного вызова для отмены и завершения на очереди. Подобным образом, если драйвер выделяет область контекста для каждого запроса для отслеживания состояния запроса, можно создать объект обратного вызова и реализовать интерфейсы обратного вызова для отмены и завершения ввода/вывода на этом объекте обратного вызова.

## Пример UMDF: объекты и интерфейсы обратных вызовов

На рис. 5.1 показаны основные интерфейсы обратных вызовов, которые образец `Fx2_Driver` реализует для объектов драйвера и устройства. На рисунке не показан интерфейс `IUnknown`, который все объекты СОМ должны поддерживать; также не показаны объекты и интерфейсы обратного вызова, поддерживающие очереди ввода/вывода драйвера.

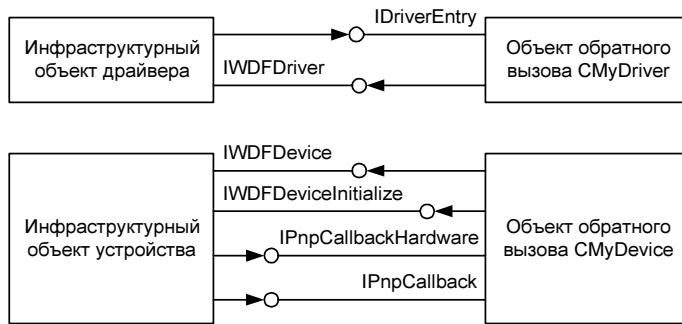


Рис. 5.1. Объекты обратного вызова драйвера и устройства в образце Fx2\_Driver

Как можно видеть на рисунке, инфраструктура реализует объект драйвера и объект устройства, а образец драйвера Fx2\_Driver создает объект обратного вызова драйвера и объект обратного вызова устройства. Инфраструктурный объект драйвера использует интерфейс `IDriverEntry` объекта обратного вызова драйвера, а объект обратного вызова драйвера, в свою очередь, использует интерфейс `IWDFDriver` на инфраструктурном объекте драйвера.

Инфраструктурный объект устройства предоставляет интерфейсы `IWDFDevice` и `IWDFDeviceInitialize`, которые используются объектом обратного вызова устройства. Объект обратного вызова устройства также реализует интерфейсы `IPnpCallbackHardware` и `IPnpCallback`, которые используются инфраструктурным объектом устройства.

## Реализация объектной модели KMDF

KMDF определяет более обширный набор объектов, чем UMDF. Причиной этому является то, что драйверы режима ядра обращаются к дополнительным ресурсам и могут выполнять определенные операции (например, обрабатывать прерывания), которые не могут выполнять драйверы пользовательского режима.

В добавок к типам объектов, поддерживаемых UMDF, KMDF также поддерживает следующие типы объектов:

- ◆ коллекции;
- ◆ строки;
- ◆ процедуры отложенного вызова (DPC);
- ◆ таймер;
- ◆ прерывания;
- ◆ интерфейс WMI;
- ◆ ассоциативный список;
- ◆ рабочий элемент;
- ◆ ключ реестра;
- ◆ несколько объектов для DMA: выключатель (enabler) DMA, транзакция DMA и общего буфера;
- ◆ несколько объектов аппаратных ресурсов: список диапазонов ресурсов, список ресурсов и список требуемых ресурсов.

KMDF не поддерживает именованное хранилище свойств; вместо этого, для управления данными постоянного хранения драйверы KMDF используют объекты ключей реестра.

Объекты WDF являются непрозрачными для драйвера, и драйвер никогда не обращается к базовой структуре напрямую. Вместо этого, драйвер KMDF обращается к объекту с помощью дескриптора. Драйвер передает дескриптор методам объекта как параметр, а KMDF передает дескриптор как параметр обратному вызову по событию.

## Типы объектов KMDF

Для драйверов KMDF инфраструктура создает объекты, для которых драйвер предоставляет указатели на функции обратного вызова. Драйверы KMDF не нуждаются в объектах обратного вызова. В табл. 5.3 перечислены все типы объектов KMDF.

**Таблица 5.3. Типы объектов KMDF**

Объект	Тип	Описание
Список потомков	WDFCHILDLIST	Представляет список дочерних устройств, которые драйвер шины перечисляет для родительского устройства
Коллекции	WDFCOLLECTION	Описывает список подобных объектов, таких как ресурсы или устройства, для которых драйвер фильтрует запросы
Устройства	WDFDEVICE	Представляет экземпляр объекта. Драйвер обычно имеет один объект WDFDEVICE для каждого управляемого им устройства
Общий блок DMA	WDFCOMMONBUFFER	Представляет буфер, к которому могут обращаться как устройство, так и драйвер для операций DMA
Выключатель DMA	WDFDMAENABLER	Разрешает использование DMA устройством. Драйвер, обрабатывающий операции ввода/вывода, имеет по одному объекту WDFDMAENABLER для каждого канала DMA в устройстве
Транзакции DMA	WDFDMA TRANSACTION	Представляет одну транзакцию DMA
Отложенный вызов процедуры (DPC)	WDFDPC	Представляет процедуру DPC
Драйвер	WDFDRIVER	Представляет сам драйвер и содержит информацию о драйвере, например о его точках входа. Каждый драйвер имеет один объект WDFDRIVER
Файл	WDFFILEOBJECT	Представляет объект файла, посредством которого внешние драйверы и приложения могут обращаться к устройству
Общий объект	WDOBJECT	Представляет общий базовый объект для использования драйвером необходимым образом
Очередь ввода/вывода	WDFQUEUE	Представляет очередь ввода/вывода. Драйвер может иметь неограниченное количество объектов WDFQUEUE
Запрос ввода/вывода	WDFREQUEST	Представляет запрос на ввод/вывод для устройства
Получатель ввода/вывода	WDFIOTARGET	Представляет стек устройства, которому драйвер направляет запрос ввода/вывода. Драйвер может иметь неограниченное количество объектов WDFIOTARGET

Таблица 5.3 (окончание)

Объект	Тип	Описание
Прерывание	WDFINTERRUPT	Представляет объект прерывания устройства. Любой драйвер, который обрабатывает прерывания устройства, имеет объект WDFINTERRUPT для каждого прерывания IRQ или MSI (Message Signaled Interrupt, прерывание с сигнальным сообщением), вызываемого устройством
Ассоциативный список	WDFLOOKASIDE	Представляет динамический список идентичных буферов, выделенных из страничного или нестраничного пула
Память	WDFMEMORY	Представляет память, используемую драйвером, обычно в виде буфера ввода или вывода, сопоставленного с запросом ввода/вывода
Ключ реестра	WDFKEY	Представляет ключ реестра
Списка ресурсов	WDFCMRESLIST	Представляет список ресурсов, фактически выделенных устройству
Список диапазона ресурсов	WDFIORESLIST	Представляет возможную конфигурацию для устройства
Список требуемых ресурсов	WDFIORESREQLIST	Представляет набор списков ресурсов ввода/вывода, в котором представлены все возможные конфигурации для устройства. Каждым элементов списка является объект WDFIORESLIST
Строка	WDFSTRING	Представляет счетную строку Unicode
Синхронизация: спин-блокировка	WDFSPINLOCK	Представляет спин-блокировку, которая используется для синхронизации доступа к данным уровня DISPATCH_LEVEL
Синхронизация: wait-блокировка	WDFWAITLOCK	Представляет wait-блокировку, которая используется для синхронизации доступа к данным уровня PASSIVE_LEVEL
Таймер	WDFTIMER	Представляет таймер, срабатывающий один раз или периодически, инициируя запуск процедуры обратного вызова
Устройство USB	WDFUSBDEVICE	Представляет устройство USB
Интерфейс USB	WDFUSBINTERFACE	Представляет интерфейс на устройстве USB
Канал USB	WDFUSBPIPE	Представляет сконфигурированный канал в установках интерфейса USB
Экземпляр WMI	WDFWMIINSTANCE	Представляет отдельный блок данных WMI, который соотнесен с определенным провайдером
Провайдер WMI	WDFWMIPROVIDER	Представляет схему для блоков данных WMI, предоставляемых драйвером
Рабочий элемент	WDFWORKITEM	Представляет рабочий элемент, который исполняется в системном потоке на уровне PASSIVE_LEVEL

### Примечание

Использование инфраструктурных объектов ограничено пределами самой инфраструктуры. Менеджер объектов Windows не управляет ими, и ими нельзя манипулировать с помощью семейства системных функций `ObXXX`. Только инфраструктура и ее драйверы могут создавать и работать с инфраструктурными объектами. Также только драйвер, создавший инфраструктурный объект, может работать с ним; работать с объектами, созданными одним драйвером, другому драйверу нельзя.

## Соглашение об именах KMDF

Имена методам, свойствам и событиям, поддерживаемым объектами KMDF, присваиваются в соответствии соглашения об именах.

Имена методам KMDF присваиваются в следующем формате:

*WdfОбъект**Операция*

Здесь:

- ◆ *Объект* указывает объект KMDF, над которым выполняет действие метод;
- ◆ *Операция* указывает действие, выполняемое методом.

Например, метод `WdfDeviceCreate` создает инфраструктурный объект устройства.

Имена свойствам KMDF присваиваются в следующем формате:

*WdfОбъект*{*Set* | *Get*}*Данные*  
*WdfОбъект*{*Assign* | *Retrieve*}*Данные*

Здесь:

- ◆ *Объект* указывает объект KMDF, над которым выполняет действие функция;
- ◆ *Данные* указывает поле для чтения или записи функцией.

Считывание и установка значения некоторых свойств всегда выполняется успешно, но для некоторых свойств эти операции могут быть неуспешными. Функции, чьи имена содержат *Set* и *Get*, всегда выполняются успешно. Функции типа *Set* возвращают `VOID`, а функции типа *Get* обычно возвращают значение поля. А функции, чьи имена содержат *Assign* и *Retrieve*, могут завершиться неуспешно, возвращая код статуса `NTSTATUS`.

Например, объект `WDFINTERRUPT` представляет объект прерывания устройства. Каждый объект прерывания описывается набором характеристик, в которых указывается тип прерывания (IRQ или MSI) и предоставляется дополнительная информация о прерывании. Эта информация считывается методом `WdfInterruptGetInfo`. Соответствующего метода для установки информации о прерывании не существует, т. к. эта информация устанавливается драйвером при создании им объекта, после чего ее нельзя изменять.

В заголовочных файлах и документации для имен заполнителей для функций KMDF обратного вызова по событию применяется следующий формат:

*Evt**Объект**Описание*

Здесь:

- ◆ *Объект* указывает объект KMDF, над которым выполняет действие функция;
- ◆ *Описание* указывает инициатора обратного вызова.

В большинстве образцов драйверов имена функциям присваиваются или ставя имя драйвера перед частью *Evt*, или же заменяя ее именем драйвера. Например, имена функций обратного вызова драйвера `Osrusbfx2` обычно начинаются с `OsrFxEvt`. Желательно придерживаться этой практики и при именовании ваших драйверов. Это улучшает удобочитаемость кода и четко указывает назначение каждой функции, при каких обстоятельствах она вызывается и какой компонент ее вызывает.

Драйвер регистрирует обратные вызовы только для событий, имеющих важность для его работы. Когда происходит событие, инфраструктура инициирует обратный вызов, передавая как параметр дескриптор объекта, для которого обратный вызов был зарегистрирован. На-

пример, если возможно извлечение устройства, его драйвер регистрирует обратный вызов *EvtDeviceEject*, который при извлечении устройства выполняет специфичные для него действия. Когда PnP-менеджер посыпает для устройства запрос IRP\_MN\_EJECT, KMDF вызывает процедуру *EvtDeviceEject* с дескриптором объекта устройства.

### Примечание

События KMDF не имеют ничего общего с событиями диспетчера ядра, предоставляемыми Windows в качестве механизма синхронизации. Драйвер не может создавать события KMDF, манипулировать ими или ожидать их. Событие KMDF — это всего лишь функция. Для ожиданий по времени KMDF предоставляет объекты таймеров.

## Создание объектов

В обеих инфраструктурах драйверы создают объекты, придерживаясь следующих постоянных принципов:

- ◆ если необходимо, драйверы UMDF создают объект обратного вызова, после чего вызывают метод для создания соответствующего инфраструктурного объекта;
- ◆ драйверы KMDF инициируют структуру атрибутов объекта и структуру конфигурации объекта, после чего вызывают метод для создания объекта.

## Создание объектов UMDF

Инфраструктура UMDF предоставляет специфичные для объекта методы создания объектов устройства, файла, запроса ввода/вывода, памяти и очереди. Чтобы создать инфраструктурный объект, драйвер UMDF выполняет следующие действия:

1. Если для инфраструктурного устройства драйвер поддерживает обратные вызовы по событию, выделяет новый экземпляр соответствующего объекта обратного вызова.
2. Создает соответствующий инфраструктурный объект и сопоставляет его объекту обратного вызова.

Драйверы UMDF выделяют объект обратного вызова, после чего создают соответствующий инфраструктурный объект посредством метода *CreateОбъект*, где *Объект* указывает тип объекта UMDF, например *Device*. Методы *CreateОбъект* также поддерживаются интерфейсами, предоставляемыми родительскими инфраструктурными объектами по умолчанию. Для методов создания объектов требуется указатель на интерфейс *IUnknown* объекта обратного вызова, чтобы инфраструктура могла определить, какой интерфейс обратного вызова поддерживается объектом обратного вызова.

В исходном коде в листинге 5.1 показано, как драйвер *Fx2\_Driver* создает объект очереди. Этот пример основан на коде из файла *ControlQueue.cpp*.

### Листинг 5.1. Создание объектов инфраструктуры и объектов обратного вызова в драйвере UMDF

```
queue = new CMYControlQueue(Device);
. . . // Код опущен.
IUnknown *callback = QueryIUnknown();
```

```
hr = m_Device->GetFxDevice()->CreateIoQueue(callback,
                                               Default,
                                               DispatchType,
                                               PowerManaged,
                                               FALSE,
                                               &fxQueue);

callback->Release();
if (FAILED(hr)) {
    . . . // Код для обработки ошибок опущен.
}
fxQueue->ReleaseQ;
```

В листинге показан базовый шаблон, применяемый драйвером UMDF для создания объекта обратного вызова и соответствующего объекта инфраструктуры. Сначала драйвер с помощью оператора `new` создает объект обратного вызова, после чего получает указатель интерфейса для объекта обратного вызова очереди. Драйвер создает инфраструктурный объект очереди с помощью метода `IWDFDevice::CreateIoQueue`, передавая ему в качестве параметров указатель интерфейса объекта обратного вызова, четыре параметра для данной очереди и указатель на переменную, в которой хранится адрес интерфейса `IWDFIoQueue` для созданного инфраструктурного объекта.

После этого драйвер освобождает ссылку, которую `QueryIUnknown` получил на интерфейс `IUnknown`, т. к. ссылку на него имеет объект устройства инфраструктуры. При условии, что создание успеха было успешным, драйвер также освобождает ссылку, которую метод `CreateIoQueue` получил на объект `fxQueue`. Дерево объектов инфраструктуры сохраняет ссылку на этот объект, поэтому в содержание дополнительной ссылки драйвером нет необходимости.

## Создание объектов KMDF

Для создания объекта KMDF драйвер выполняет такую последовательность операций:

1. Если имеется структура конфигурации для объекта, инициализирует ее.
2. Если необходимо, инициализирует структуру атрибутов объекта.
3. Создает объект, вызывая для этого метод создания объекта.

## Структура конфигурации объекта KMDF

KMDF определяет структуры конфигурации для большинства объектов. Этим структурам присваиваются имена в формате `WDF_Object_CONFIG`. Структура конфигурации объекта содержит указатели на специфичную для объекта информацию, например, функции драйвера обратного вызова по событию для объекта. Драйвер заполняет эту инфраструктуру, после чего передает ее инфраструктуре, когда он вызывает метод создания объекта. Инфраструктура использует информацию в структуре конфигурации для инициализации объекта. Например, объекту `WDF_DRIVER` требуется указатель на функцию обратного вызова `EvtDriverDeviceAdd` драйвера, которую KMDF инициализирует, когда происходит событие Plug and Play добавления устройства.

Для инициализации структур конфигурации в KMDF определены функции, именуемые в формате `WDF_Object_CONFIG_INIT`, где `Object` означает название типа устройства. Не все типы объектов имеют структуры конфигурации или соответствующие функции инициализации.

## Структура атрибутов объекта KMDF

Каждый объект KMDF имеет набор атрибутов, которые перечислены в табл. 5.4. При создании драйвером KMDF-объекта, драйвер передает указатель на структуру атрибутов, в которой содержатся значения атрибутов объекта. Не все атрибуты используются всеми объектами.

**Таблица 5.4. Атрибуты объекта KMDF**

Поле	Описание
ContextSizeOverride	Размер области контекста, который превалирует над значением в ContextTypeInfo>ContextSize. Это поле может быть полезным при работе с областями контекста, которые имеют разные размеры
ContextTypeInfo	Указатель на информацию о типе для области контекста объекта
EvtCleanupCallback	Указатель на процедуру обратного вызова, запускаемую для очистки ресурсов объекта перед его удалением. После выполнения, все еще могут оставаться открытые ссылки на объект
EvtDestroyCallback	Указатель на процедуру обратного вызова, запускаемую, когда значение счетчика ссылок для объекта, обозначенного для удаления, достигает нуля
ExecutionLevel	Максимальный уровень IRQL, при котором KMDF может активировать некоторые обратные вызовы
ParentObject	Дескриптор родителя объекта
Size	Размер структуры в байтах
SynchronizationScope	Уровень, при котором синхронизируются определенные обратные вызовы для данного объекта. Это поле применимо только к объектам драйвера, устройства и файла

Драйвер предоставляет атрибуты в структуре `WDF_OBJECT_ATTRIBUTES`. Для инициализации этой структуры KMDF определяет следующие функции и макросы:

- ◆ `WDF_OBJECT_ATTRIBUTES_INIT`;
- ◆ `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE`;
- ◆ `WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE`.

Функция `WDF_OBJECT_ATTRIBUTES_INIT` устанавливает значения уровней синхронизации и исполнения, которые определяют, какие обратные вызовы драйвера инфраструктура KMDF активирует параллельно и самый высший уровень IRQL, при котором они могут быть вызваны. По умолчанию значения обоих этих атрибутов устанавливаются такими же, как и родительского объекта. Следующие два макроя инициализируют тип контекста, устанавливая информацию об области контекста объекта. (Область контекста объекта описывается в одноименном разделе дальше в этой главе.)

Хотя атрибуты можно установить для любого типа устройств, для большинства объектов обычно достаточно установок по умолчанию. Чтобы принять установки по умолчанию, драйвер указывает функцию `WDF_NO_OBJECT_ATTRIBUTES`, которую WDF определяет как `NULL`. Но если драйвер определяет область контекста для объекта, он также должен указать атрибуты, т. к. размер и тип области контекста указываются в структуре атрибутов.

## Методы создания объектов KMDF

После инициализации структур конфигурации и атрибутов драйвер создает объект. Делает он это, вызывая метод `WdfObjectCreate`, где `Object` указывает тип объекта.

Метод создания объекта возвращает дескриптор созданного объекта, который впоследствии используется драйвером для обращения к объекту. Методы создания объектов выполняют следующие операции:

- ◆ выделяют для объекта и его области контекста память из нестраничного пула;
- ◆ инициализируют атрибуты объекта значениями по умолчанию и, если необходимо, значениями, указанными в технических требованиях драйвера;
- ◆ обнуляют области контекста объекта;
- ◆ конфигурируют объект, сохраняя указатели на его обратные вызовы по событию и устанавливая другие специфичные для устройства характеристики.

В случае неуспешной инициализации инфраструктура удаляет объект.

В листинге 5.2 показан исходный код для создания объекта в драйвере KMDF. Данный пример взят из файла исходного кода `Device.c` для драйвера `Osrusbfx2`.

### Листинг 5.2. Создания объекта в драйвере KMDF

```
NTSTATUS status = STATUS_SUCCESS;
WDFDEVICE device;
WDF_IO_QUEUE_CONFIG ioQueueConfig;
WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig, WdfIoQueueDispatchSequential);
ioQueueConfig.EvtIoRead = OsrFxEvtIoRead;
ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;
status = WdfIoQueueCreate( device,
                           &ioQueueConfig,
                           WDF_NO_OBJ ECT_ATTRIBUTES,
                           &queue // Дескриптор очереди.
                     );
if (!NT_SUCCESS (status)) {
    . . . // Код для обработки ошибок опущен.
}
```

В листинге 5.2 показано, как драйвер `Osrusbfx2` создает объект очереди. В следующем описании процесса создания драйвера основное внимание уделяется базовым принципам, опуская специфичные для очереди подробности.

Драйвер инициализирует структуру конфигурации объекта очереди, вызывая функцию `WDF_IO_QUEUE_CONFIG_INIT`, передавая ей указатель на структуру и специфичный для очереди параметр, который описывает, каким образом очередь планирует обработку запросов. После этого в двух дополнительных полях структуры конфигурации драйвер регистрирует обратные вызовы по событию для объекта очереди. Для данного объекта очереди драйвер не меняет родителя по умолчанию и не устанавливает какие-либо другие атрибуты объекта, поэтому необходимо инициализировать структуру атрибутов объекта.

Наконец, драйвер создает объект очереди, вызывая для этого метод `WdfIoQueueCreate`. Функции передаются следующие четыре параметра:

- ◆ дескриптор инфраструктурного объекта устройства, сопоставленного очереди;
- ◆ указатель на инициализированную структуру конфигурации очереди;

- ◆ константу WDF\_NO\_OBJECT\_ATTRIBUTES для хранения атрибутов по умолчанию;
- ◆ адрес, по какому метод возвращает дескриптор для очереди.

### Совет

В главе 24 рассматривается вопрос, каким образом необходимо вставлять примечания в функции обратного вызова драйвера, чтобы SDV мог анализировать их на соответствие правилам KMDF. Согласно требованиям этих правил, перед созданием объекта устройства драйвер всегда должен вызывать интерфейсы DDI для инициализации объекта устройства.

## Иерархия и время жизни объектов

Как UMDF, так и KMDF управляют временем жизни объектов, ведя учет количеству ссылок на объект. Для каждого объекта ведется подсчет ссылок на него, что позволяет обеспечить существование объекта все время, когда он используется. Применяются следующие методы подсчета ссылок:

- ◆ UMDF использует стандартный интерфейс COM. Для управления подсчетом ссылок на объект драйверы могут применять к нему методы AddRef и Release;
- ◆ KMDF ведет подсчет ссылок на объект сама. Драйверы могут применять методы WdfObjectReferenceXxx и WdfObjectDereferenceXxx.

Кроме подсчета ссылок на объект, за исключением объекта драйвера, при создании каждому инфраструктурному объекту назначается родитель. Объект драйвера является корневым объектом; все другие объекты иерархии являются его подчиненными. Объекты устройства всегда являются потомками объекта драйвера, а объекты очереди обычно являются потомками объектов устройства, хотя очередь может быть и более поздним потомком, т. е. ее родитель может находиться в иерархии ниже объекта устройства. Инфраструктура создает и поддерживает внутренне дерево объектов, основанное на этих взаимоотношениях. Простой пример такого дерева объектов показан на рис. 5.2.

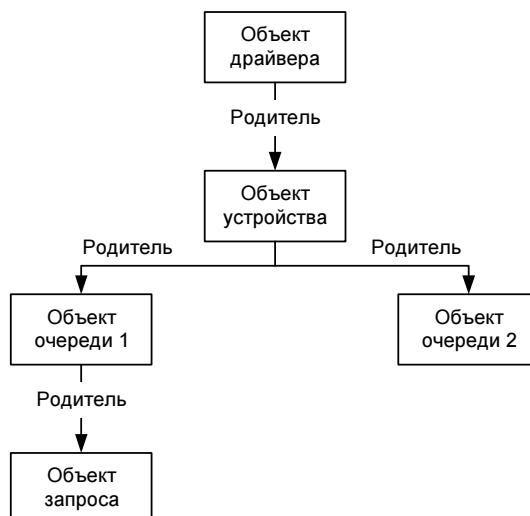


Рис. 5.2. Пример дерева объектов

Из рисунка видно, что объект драйвера является родителем объекта устройства. Объект устройства является родителем каждого из двух объектов очереди. Объект очереди 1 является родителем объекта запроса. А объект очереди 1, объект устройства и объект драйвера — все являются предками объекта запроса.

### Совет

Для драйверов, создающих объекты, необходимо обязательно установить родителей этих объектов должным образом. Рассмотрим, например, объект памяти. По умолчанию родителем объекта памяти является объект драйвера. Такое взаимоотношение адекватно для памяти, которую драйвер выделяет для всеобщего использования драйвером, т. к. по умолчанию объект памяти существует до тех пор, пока не удален объект драйвера. Но если объект памяти используется только в определенных типах запросов ввода/вывода, более подходящим родителем для него был бы сам объект запроса или получатель ввода/вывода, которому драйвер посыпает эти запросы. Решающим фактором в таких случаях является длительность использования объекта памяти. Если драйвер создает новый объект памяти для каждого запроса, его родителем необходимо установить запрос, чтобы объект памяти можно было удалить по завершению запроса. А если один и тот же объект памяти используется драйвером для последовательных запросов к одному и тому же получателю ввода/вывода, то тогда его родителем нужно установить этот получатель ввода/вывода.

При удалении объекта инфраструктура вызывает функции обратного вызова зачистки ресурсов, начиная с его самых дальних потомков и продвигаясь вверх по иерархии до самого объекта. Например, при удалении объекта устройства на рис. 5.2 инфраструктура сначала вызывает функцию обратного вызова зачистки ресурсов объекта запроса, потом объектов очередей и, наконец, самого объекта устройства.

Взаимоотношения "родитель — потомок" играют важную роль в упрощении очистки ресурсов объекта и управления состоянием. Рассмотрим следующий пример: объект по умолчанию получателя ввода/вывода является потомком объекта устройства. При удалении объекта устройства инфраструктура сначала удаляет объект получателя ввода/вывода и вызывает функцию обратного вызова очистки ресурсов получателя ввода/вывода. Таким образом, драйверу не требуется специальный код для управления запросами ввода/вывода, которые завершаются во время ликвидации объекта устройства. Такие задачи обычно решаются функцией обратного вызова получателя ввода/вывода.

Драйверы обычно не получают ссылок на создаваемые ими объекты, но в некоторых случаях это необходимо для того, чтобы обеспечить дляящуюся действительность дескриптора объекта. Например, драйвер, посылающий асинхронный запрос ввода/вывода, может получить ссылку на объект запроса с тем, чтобы помочь предотвратить возникновение состояния гонок во время отмены. Драйвер должен освободить эту ссылку, прежде чем он может удалить этот объект запроса. Для освобождения такой ссылки драйвер должен реализовать обратный вызов очистки ресурсов объекта. Для получения дополнительной информации по этому вопросу см. разд. "Обратные вызовы очистки ресурсов" далее в этой главе.

### Удаление, ликвидация, очистка и уничтожение

Разобраться со значениями терминов "удаление", "ликвидация", "очистка" и "уничтожение" иногда может быть непросто. Когда драйвер вызывает инфраструктуру, чтобы удалить объект, инфраструктура начинает выполнение последовательности операций, конечным результатом которых будет уничтожение объекта. Первым шагом в этой последовательности является ликвидация всех потомков объекта, для чего вызываются их функции обратного

вызыва очистки ресурсов. Исполнение этого шага начинается с наиболее удаленного в иерархии от родителя поколения и продолжается в строгом иерархическом порядке.

После завершения ликвидации могут оставаться ссылки на какого-либо потомка или на родителя. После удаления последней ссылки на ликвидируемый объект инфраструктура выполняет обратный вызов функции удаления для объекта и потом уничтожает объект.

## Иерархия объектов UMDF

Иерархия взаимоотношений "родитель — потомок" для объектов UMDF показана на рис 5.3.

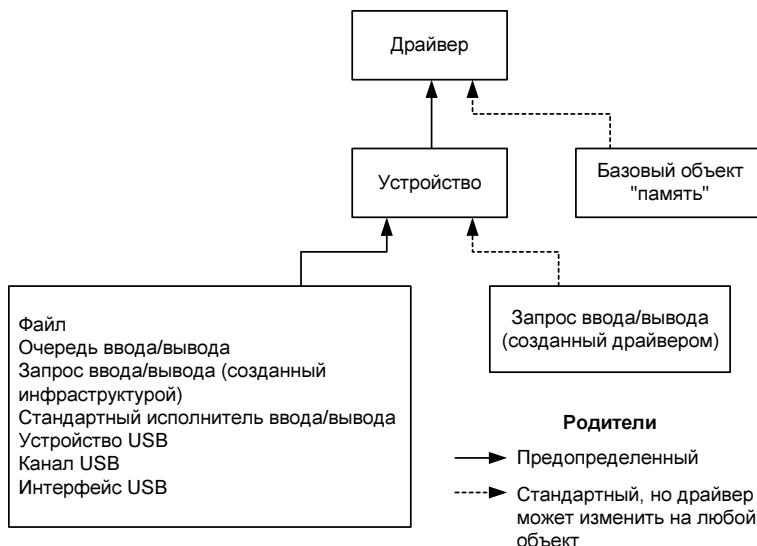


Рис. 5.3. Иерархия взаимоотношений "родитель — потомок" для объектов UMDF

Как можно видеть на рисунке, объект драйвера является корневым объектом, а объекты устройств являются его потомками. По умолчанию созданные драйвером объекты памяти и базовые объекты также являются потомками объекта драйвера, но драйвер может изменить родителя объекта памяти или базового объекта на другой объект. Например, драйвер может изменить родителя объекта памяти на созданный драйвером объект запроса ввода/вывода, с которым используется данный объект памяти.

Объект устройства является родителем по умолчанию для всех инфраструктурных объектов. Драйвер может изменить родителя создаваемого им объекта запроса ввода/вывода при создании запроса. Если запрос имеет близкое отношение к объекту запроса ввода/вывода, созданного инфраструктурой, драйвер должен установить родителем данный инфраструктурный объект запроса. Например, если драйверу требуется получить информацию от другого стека устройства или послать запрос IOCTL вниз по своему стеку, прежде чем он может завершить запрос ввода/вывода, полученный от инфраструктуры, драйвер должен установить в качестве родителя этот созданный инфраструктурой объект. Когда удаляется запрос, созданный инфраструктурой, также удаляется и запрос, созданный драйвером.

Время жизни объекта обратного вызова связано со временем жизни соответствующего инфраструктурного объекта. Когда драйвер создает инфраструктурный объект, он передает

указатель на интерфейс на соответствующем объекте обратного вызова. Инфраструктурный объект получает ссылку на объект обратного вызова. Например, когда драйвер вызывает метод `IWDFDriver::CreateDevice`, он передает указатель на интерфейс на объекте обратного вызова данного устройства. Инфраструктура создает инфраструктурный объект устройства, получает ссылку на объект обратного вызова и возвращает указатель на интерфейс `IWDFDevice`.

Подсчет ссылок позволяет обеспечить существование объекта все время, когда он используется. Если счет ссылок становится нулевым, объект можно удалить. Драйвер удаляет объекты с помощью метода `IWDFObject::DeleteWdfObject`.

UMDF получает ссылки по правилам СОМ. Соответственно, всякий раз, когда метод UMDF возвращает указатель интерфейса, как выходной параметр, инфраструктура получает ссылку на этот интерфейс. Когда драйвер больше не использует указатель, он должен освободить эту ссылку с помощью метода `Release`.

Для увеличения или уменьшения значения счетчика ссылок на объект в коде драйвера можно применить методы `AddRef` и `Release` интерфейса `IUnknown`, но обычно в этом нет необходимости. Но если драйвер все же получает ссылку на инфраструктурный объект, он должен реализовать интерфейс `IObjectCleanup::OnCleanup` на соответствующем объекте функции обратного вызова, чтобы удалить эту ссылку, с тем, чтобы инфраструктура могла в должное время удалить объект. Инфраструктура вызывает этот метод при удалении объекта.

Программистам, работающим с СОМ, наверное, приходилось встречаться с циклическими ссылками, когда два объекта имеют ссылки друг на друга. Хотя каждый объект освобождает свои ссылки на другой объект, когда счетчик ссылок на него самого становится нулевым, ни один из объектов нельзя удалить самостоятельно. Внешнее событие должно заставить один из этих двух объектов освободить свою ссылку на другой объект, таким образом, разрывая циклическую ссылку и позволяя обоим объектам удалить себя. Это и является главным назначением обратного вызова `OnCleanup`.

Возникновения циклических ссылок в драйвере очень часто можно избежать. Для этого объект обратного вызова может хранить слабую ссылку на инфраструктурный объект, т. е. ссылку, которая не учитывается при подсчете ссылок на объект. Прежде чем разрушить этот инфраструктурный объект, инфраструктура вызывает метод `OnCleanup`, который просто сбрасывает указатель и прекращает его использование.

Подробное объяснение, когда использовать методы `AddRef` и `OnCleanup`, дается в главе 18.

## Иерархия объектов KMDF

Корневым объектом драйверов KMDF является объект драйвера, все остальные объекты являются его потомками. Для некоторых типов объектов, при создании объекта драйвер может указать его родителя. Если при создании объекта драйвер не указывает его родителя, инфраструктура устанавливает родителем по умолчанию объект драйвера.

Иерархия по умолчанию объектов KMDF показана на рис. 5.4.

На рис. 5.4 показано, какие другие объекты должны быть в цепочке родителей каждого объекта. Эти объекты не обязательно должны быть непосредственными родителями, но могут быть удаленными на одно, два или больше поколений. Например, на рис. 5.4 показано, что объект `WDFDEVICE` является родителем объекта `WDFQUEUE`. Но объект `WDFQUEUE` мог быть потомком объекта `WDFIOTARGET`, который в свою очередь является потомком объекта `WDFDEVICE`. Таким образом, объект `WDFDEVICE` находится в цепи родителей объекта `WDFQUEUE`.

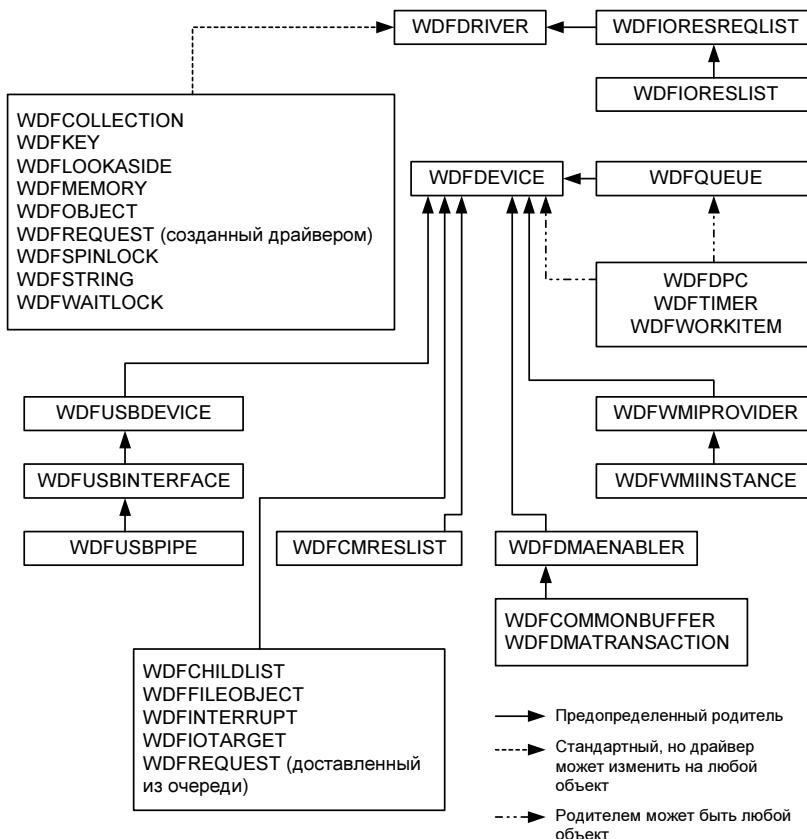


Рис. 5.4. Иерархия взаимоотношений "родитель — потомок" для объектов KMDF

KMDF косвенно ведет счет ссылок для каждого объекта и обеспечивает существование объекта до тех пор, пока не освободятся все ссылки на него. Если драйвер явно удаляет объект вызовом метода удаления, KMDF помечает объект на удаление, но не освобождает память объекта до тех пор, пока счетчик ссылок на объект не уменьшится до нуля.

Хотя драйверы KMDF не часто получают прямые ссылки, если драйверу необходимо получить прямую ссылку, инфраструктура предоставляет ему для этих целей методы `WdfObjectReference` и `WdfObjectDereference`. Драйвер должен получить прямую ссылку, когда необходимо обеспечить длительную достоверность дескриптора объекта.

Драйвер должен освободить эту ссылку, прежде чем он сможет удалить этот объект. Драйвер освобождает такую ссылку, вызывая метод `wdfObjectDereference` из обратного вызова очистки ресурсов.

## Удаление объектов

Удаление объектов может быть инициировано любым из следующих действий:

- ◆ удаление родителя объекта;
- ◆ вызов драйвером метода, который явно удаляет объект:
  - для драйверов UMDF это будет метод `DFOBJECT::DeleteWdfObject`;
  - для драйверов KMDF применяется метод `WdfObjectDelete`.

Удалением большинства объектов управляет инфраструктура. Драйверы могут вызывать методы удаления только для объектов, которыми они владеют и управляют. В табл. 5.5 перечислены типы инфраструктурных объектов, с указанием для каждого типа объекта, может ли он быть удален драйвером.

**Таблица 5.5. Типы объектов, которые могут быть удалены драйвером**

Объект	UMDF	KMDF
Список потомков	Неприменимо	Нет
Коллекция	Неприменимо	Да
Устройство	Нет	Нет, за исключением объектов устройств управления и объектов PDO в определенном состоянии
Общий буфер DMA	Неприменимо	Да
Выключатель DMA	Неприменимо	Да
Транзакция DMA	Неприменимо	Да
Отложенный вызов процедуры (DPC)	Неприменимо	Да
Драйвер	Нет	Нет
Созданный драйвером файл	Да	Да
Файл	Нет	Нет
Базовый или общий объект	Да, если созданный драйвером	Да, если созданный драйвером
Очередь ввода/вывода	Нет — для очереди ввода/вывода по умолчанию. Да — для других очередей	Да
Запрос ввода/вывода	Нет, если созданный инфраструктурой. Да, если созданный драйвером	Нет, если созданный инфраструктурой. Да, если созданный драйвером
Получатель ввода/вывода	Нет — для получателя ввода/вывода по умолчанию. Да — для всех других получателей	Нет — для получателя ввода/вывода по умолчанию. Да — для всех других получателей
Прерывание	Неприменимо	Нет
Ассоциативный список	Неприменимо	Да
Память	Нет, если созданный инфраструктурой. Да, если созданный драйвером	Нет, если созданный инфраструктурой. Да, если созданный драйвером
Именованное хранилище свойства	Нет	Неприменимо
Ключ реестра	Неприменимо	Да
Список ресурсов	Неприменимо	Нет
Список диапазона ресурсов	Неприменимо	Да

Таблица 5.5 (окончание)

Объект	UMDF	KMDF
Список требуемых ресурсов	Неприменимо	Нет
Строка	Неприменимо	Да
Синхронизация: спин-блокировка	Неприменимо	Да
Синхронизация: wait-блокировка	Неприменимо	Да
Таймер	Неприменимо	Да
Устройство USB	Нет — для получателя ввода/вывода по умолчанию. Да — для всех других получателей	Да
Интерфейс USB	Нет — для получателя ввода/вывода по умолчанию. Да — для всех других получателей	Нет
Канал USB	Нет — для получателя ввода/вывода по умолчанию. Да — для всех других получателей	Нет
Экземпляр WMI	Неприменимо	Да
Провайдер WMI	Неприменимо	Нет
Рабочий элемент	Неприменимо	Да

Инфраструктура владеет большинством инфраструктурных объектов и управляет их временным жизнью. Например, инфраструктура управляет объектом устройства и стандартным объектом получателя ввода/вывода. Когда устройство удалено из системы, инфраструктура удаляет объект устройства и стандартный объект получателя ввода/вывода.

Если владельцем объекта является драйвер, то когда объект больше не нужен, драйвер должен явно удалить его. Когда драйвер вызывает метод удаления объекта, инфраструктура выполняет следующие действия:

- ◆ освобождает свои ссылки на интерфейсы объекта обратного вызова (применимо только в UMDF);
- ◆ независимо от состояния счетчика ссылок на объект, сразу же активирует обратный вызов очистки объекта;
- ◆ уменьшает значение внутреннего счетчика ссылок на инфраструктурный объект;
- ◆ помечает инфраструктурный объект на удаление;
- ◆ когда значение счетчика ссылок падает до нуля, удаляет инфраструктурный объект.

Если на объект остаются ссылки, инфраструктура помечает объект на удаление, но не удаляет его до тех пор, пока счетчик ссылок на него не примет нулевое значение. Таким образом, инфраструктура обеспечивает, что значение счетчика ссылок на объект остается ненулевым, а объект, соответственно, действительным, на протяжении всего времени, когда он может понадобиться. Когда значение счетчика ссылок на объект падает до нуля, инфра-

структура активирует обратный вызов удаления, который драйвер зарегистрировал для данного объекта, и удаляет объект с внутреннего дерева объектов.

Если драйвер не удаляет объект явно, инфраструктура удаляет его, когда удаляется родитель объекта.

Драйверу нельзя вызывать метод удаления для объекта, контролируемый инфраструктурой, т. к.:

- ◆ в драйвере UMDF это действие вызывает сбой;
- ◆ в драйвере KMDF это вызывает останов bugcheck.

## Обратные вызовы очистки ресурсов

Драйвер может поддерживать обратный вызов очистки для любого объекта. Инфраструктура вызывает обратный вызов зачистки перед удалением объекта, прямого родителя объекта или любого из более дальних родителей объекта. Перед тем как удалить объект, обратный вызов очистки должен освободить все существующие ссылки, полученные объектом, освободить все ресурсы, выделенные драйвером от имени объекта, а также выполнить все другие необходимые задачи.

Драйверы UMDF поддерживают обратный вызов очистки посредством интерфейса `IObjectCleanup` на объекте обратного вызова. Этот интерфейс имеет лишь один метод — `OnCleanup`. Инфраструктура вызывает метод `OnCleanup` как часть последовательности разрушения объекта перед освобождением объекта обратного вызова.

Инфраструктура KMDF определяет обратный вызов `EvtCleanupCallback`. Инфраструктура вызывает функцию обратного вызова `EvtCleanupCallback` при удалении объекта, но перед тем, как значение его счетчика ссылок примет нулевое значение. Драйвер сохраняет указатель на эту функцию в структуре атрибутов объекта при его создании.

## Обратные вызовы деструкции

Вдобавок к обратному вызову очистки, или вместо него, драйвер может создать для объекта обратный вызов деструкции.

Инфраструктура UMDF не определяет обратный вызов деструкции. Когда счетчик ссылок на объект обратного вызова принимает нулевое значение, объект обычно инициирует свой деструктор, роль которого исполняет обратный вызов деструкции. С помощью деструктора драйвер может очистить ресурсы, которые он выделил от имени объекта обратного вызова. Но драйвер не может отслеживать уничтожение инфраструктурного объекта. Удаление инфраструктурного объекта и объекта обратного вызова не обязательно происходит одновременно. Если драйвер или другие инфраструктурные объекты все еще имеют ссылки на объект обратного вызова, этот объект может быть удален некоторое время после удаления инфраструктурного объекта.

Для драйвера KMDF функцией деструктора является обратный вызов `EvtDestroyCallback`, который драйвер регистрирует в структуре атрибутов объекта при создании объекта. Инфраструктура вызывает функцию деструктора после того, как счетчик ссылок на объект примет нулевое значение и после завершения обратных вызовов очистки прямого и более удаленных родителей объекта. Инфраструктура выполняет фактическое удаление объекта, когда обратный вызов деструкции возвращает управление.

## Удаление объектов UMDF

Большинство объектов, используемых драйвером UMDF, удаляются инфраструктурой. Когда устройство удаляется из системы, инфраструктура удаляет объект устройства и всех его потомков. Если драйвер устанавливает соответствующего родителя для создаваемых им объектов, инфраструктура удаляет эти объекты, когда она удаляет родителя. Например, когда драйвер создает объект памяти для отправки в запросе ввода/вывода, он должен установить родителем этого объекта данный запрос ввода/вывода. По завершению запроса ввода/вывода инфраструктура удаляет объект памяти.

Создаваемый драйвером запрос ввода/вывода является наиболее распространенным объектом, который может быть явно удален драйвером UMDF. Например, драйвер Fx2\_Driver создает и посыпает запросы ввода/вывода, а потом удаляет объект запроса, активируя обратный вызов завершения ввода/вывода. В листинге 5.3 показано, каким образом драйвер вызывает метод `IWDFObject::DeleteWdfObject` из своего метода `IWDFRequestCallbackRequestCompletion::OnCompletion` в файле `Device.cpp`.

### Листинг 5.3. Удаление объекта драйвером UMDF

```
VOID CMyDevice::OnCompletion(
    IWDFIoRequest*     FxRequest,
    IWDFIoTarget*      pIoTarget,
    IWDFRequestCompletionParams* pParams,
    PVOID              pContext
)
{
    . . . // Код опущен.

    HRESULT hrCompletion = pParams->GetCompletionStatus();
    if (FAILED(hrCompletion)) {
        m_InterruptReadProblem = hrCompletion;
    }
    else {
        // Получаем возвращенные параметры и другую информацию.
        . . . // Код опущен.
    }
    FxRequest->DeleteWdfObject();
    . . . // Код опущен.
}
```

В обратном вызове завершения драйвер получает всю требуемую ему от объекта запроса информацию перед вызовом метода `DeleteWdfObject`. После возвращения управления от `DeleteWdfObject` драйвер больше не может получить доступ к объекту.

## Удаление объектов KMDF

При удалении объекта KMDF всего его потомки также удаляются. Удаление объектов начинается в самого далекого от объекта потомка и продвигается вверх по иерархии объектов, пока не будет удален сам родительский объект.

Рассмотрим процесс на примере удаления объекта, называющегося `YourObject`. Для его удаления инфраструктура выполняет такую последовательность действий:

1. Вызывает функцию `EvtCleanupCallback` для самого удаленного в инфраструктурном дереве объектов потомка объекта.

Прежде чем удалить родителей любого объекта, для него необходимо выполнить все требуемые задачи по очистке ресурсов, для чего вызывается его функция `EvtCleanupCallback`. Такие задачи очистки могут включать освобождение явных ссылок на сам объект или на его родительский объект. Во время исполнения функции `EvtCleanupCallback` объекта его дочерние объекты все еще существуют, хотя их функции `EvtCleanupCallback` уже и были вызваны.

2. Повторяет шаг 1 для каждого потомка, двигаясь вверх по дереву иерархии и заканчивая самим объектом `YourObject`.
3. Обходит дерево в том же порядке снова, для того чтобы удалить удерживаемые структурой ссылки на объект.

Если в результате счетчик ссылок примет нулевое значение, инфраструктура вызывает функцию `EvtDestroyCallback` объекта, а потом освобождает память, выделенную объекту и его области контекста.

Инфраструктура обеспечивает вызов функций очистки для потомков перед вызовом функций очистки их родительских объектов. При этом, однако, вызов функций очистки для элементов одного уровня выполняется в произвольном порядке. Например, если объект устройства имеет три дочерних объекта очереди, то инфраструктура вызывает функции очистки для этих объектов в любом порядке.

### ***О реализации очистки и удаления объектов в KMDF***

С первого взгляда, реализация модели удаления объектов, описанной в этой главе, казалась простой задачей, но в действительности сделать это оказалось на удивление сложно. Но, сделав мысленный шаг назад и мысленно охватив взглядом всю схему, можно видеть, что удаление является транзакцией состояния, охватывающей несколько объектов. Любой объект может находиться в середине процесса исполнения той же самой транзакции, которая только инициируется для одного из его родителей, как, например, при удалении родительского объекта в одном потоке и дочернего объекта в другом потоке. Каждый объект должен быть способен переходить от состояния существования в состояние ликвидации, а также предотвращать состояния гонок. Например, инфраструктура может одновременно ликвидировать объект и его родителя, или драйвер клиента может явно удалить объект в то же самое время, как инфраструктура удаляет родителя этого же объекта, таким образом, косвенно удаляя сам объект.

Подобно многим проблемам в инфраструктуре, мы решили эту проблему, реализуя простую машину состояний и формализуя разные вводы, которые воздействуют на время жизни объекта. На этом этапе мы знали, каким образом ликвидировать дерево, но нам еще нужно было придумать способ, как определять завершение удаления. Нашей целью было сделать процесс удаления и выключения устройства как можно легким для драйвера, не прибегая к синхронизации, обычно требуемой драйверами WDM. В KMDF ликвидация объекта должна происходить на уровне объекта устройства, с тем, чтобы можно было гарантировать уничтожение инфраструктурой всех потомков перед уничтожением самого объекта устройства и уничтожение всех объектов в области видимости драйвера перед разрушением объекта драйвера. Мы решили эту проблему, помещая каждый объект в список предназначенных для удаления объектов, специфичных для устройства или находящихся в области видимости драйвера. Инфраструктура проверяет список, чтобы обеспечить удаление объектов в соответствующее время. (Даун Холэн (*Down Holan*), команда разработчиков Windows Driver Foundation, Microsoft.)

**Порядок вызова инфраструктурой KMDF функций `EvtCleanupCallback`.** Применение драйвером иерархии объектов инфраструктуры для удаления дочерних объектов гарантирует, что когда исполняется функция `EvtCleanupCallback` для дочернего объекта, родительский объект все еще существует. Исключением этому является следующий случай — когда драй-

вер явно удаляет объект с помощью метода `WdfObjectDelete`, гарантий, что родительский объект все еще существует, нет.

Для некоторых типов объектов KMDF может вызывать функции `EvtCleanupCallback` на IRQL-уровне PASSIVE\_LEVEL или DISPATCH\_LEVEL. Но для перечисленных далее объектов инфраструктура всегда вызывает функции `EvtCleanupCallback` только на уровне PASSIVE\_LEVEL.

- ◆ WDFDEVICE;
- ◆ WDFQUEUE;
- ◆ WDFDPC;
- ◆ WDFSTRING;
- ◆ WDFIOTARGET;
- ◆ WDFTIMER;
- ◆ объекты WDFLOOKASIDE, WDFWORKITEM, выделяющие память из страничного пула;
- ◆ объекты WDFMEMORY, содержащие память из страничного пула.

Если объект должен ожидать завершения какой-либо операции или обращается к страничной памяти, очистка для него должна выполняться на уровне IRQL PASSIVE\_LEVEL. Если драйвер пытается удалить такой объект, или родителя такого объекта, на уровне DISPATCH\_LEVEL, то KMDF откладывает ликвидацию всего дерева и организовывает очередь обратных вызовов функций очистки для рабочего элемента для обработки на уровне PASSIVE\_LEVEL позже. Таким образом, обратный вызов очистки для родителя исполняется после выполнения обратных вызовов очистки для всех потомков. В настоящее время KMDF содержит одну очередь рабочего элемента для каждого объекта устройства и, когда требуется, вызывает процедуры для объекта устройства и его потомков посредством этой очереди.

Объекты, требующие выполнения очистки на уровне PASSIVE\_LEVEL, не обязательно создавать на уровне PASSIVE\_LEVEL.

Объекты WDFWORKITEM, WDFTIMER, WDFDPC и WDFQUEUE можно создавать на уровне DISPATCH\_LEVEL; на этом же уровне можно также создавать объекты WDFMEMORY и WDFLOOKASIDE, которые используют память из нестраничного пула.

Объекты WDFMEMORY и WDFLOOKASIDE, использующие страничный пул, и объекты WDFDEVICE, WDFIOTARGET, WDFCOMMONBUFFER, WDFKEY, WDFCHILDLIST и WDFSTRING на уровне DISPATCH\_LEVEL создавать нельзя.

### **Исключение: завершенные запросы ввода/вывода**

Исключением к гарантированному существованию родителя при исполнении обратных вызовов очистки потомков являются запросы ввода/вывода, завершающиеся на уровне DISPATCH\_LEVEL. Если такой объект запроса ввода/вывода имеет один или больше дочерних объектов, чьи процедуры `EvtCleanupCallback` должны вызываться на уровне PASSIVE\_LEVEL, родительский объект запроса ввода/вывода можно удалить до удаления его потомков.

Рассмотрим, что происходит при завершении драйвером запроса ввода/вывода, у которого есть дочерний объект таймера. В соответствии с контрактом для объекта запроса ввода/вывода, KMDF вызывает процедуру `EvtCleanupCallback` объекта запроса, пока указатель

на нижележащий пакет IRP WDM все еще является действительным и доступным. Для большей производительности, KMDF должна завершить обработку пакета IRP как можно быстрее. Но как только обработка пакета IRP завершена, указатель на пакет IRP больше не действителен. Вместо того чтобы ожидать завершения исполнения процедур *EvtCleanupCallback* на уровне PASSIVE\_LEVEL для завершения пакета IRP, KMDF ставит обратный вызов очистки таймера в очередь к рабочему элементу, а потом завершает обработку пакета IRP. Поэтому функция *EvtCleanupCallback* для таймера может получаться после того, как запрос был завершен, указатель на пакет IRP освобожден и объект запроса удален.

Во избежание каких-либо проблем, которые могли бы возникнуть вследствие такого поведения, драйверы не должны устанавливать объект запроса ввода/вывода родителем любого объекта, требующего очистки на уровне PASSIVE\_LEVEL.

**Порядок вызова инфраструктурой KMDF функций *EvtDestroyCallback*.** KMDF вызывает процедуру *EvtDestroyCallback* для объекта после вызова его процедуры *EvtCleanupCallback* и после того, как счетчик ссылок на объект примет нулевое значение. Процедуры *EvtCleanupCallback* для объекта и его потомков можно вызывать в любом порядке, включая вызов процедуры *EvtCleanupCallback* для родителя перед вызовом для потомка. Процедура *EvtDestroyCallback* может обращаться к области контекста объекта, но не может вызывать никаких методов объекта.

Инфраструктура не гарантирует уровень IRQL, на котором она вызывает процедуру *EvtDestroyCallback* объекта.

**Пример удаления объекта KMDF.** В листинге 5.4 показан простой пример удаления объекта из файла Driver.c образца драйвера Echo.

#### Листинг 5.4. Удаление объекта KMDF

```
NTSTATUS status;
WDFSTRINC string;
WDF_DRIVER_VERSION_AVAILABLE_PARAMS ver;
status = WdfStringCreate(NULL, WDF_NO_OBJECT_ATTRIBUTES, &string);
if (!NT_SUCCESS(status)) {
    . . . // Код опущен.
}
status = WdfDriverRetrieveVersionString(WdfCetDriver(), string);
if (!NT_SUCCESS(status)) {
    . . . // Код опущен.
}
. . . // Код опущен.
WdfObj ectDelete(string);
string = NULL; // Чтобы предотвратить ссылку на удаленный объект.
```

В данном примере драйвер Echo создает объект строки для передачи методу, который извлекает версию драйвера. После окончания работы с объектом драйвер вызывает метод *WdfObjectDelete*.

## Область контекста объекта

Область контекста объекта представляет собой определенную драйвером область памяти для данных, которые драйвер использует с определенным объектом. Драйверы UMDF и KMDF обычно используют свои области контекста объекта каждый по-своему.

А именно:

- ◆ драйверы UMDF обычно сохраняют данные контекста в объекте обратного вызова драйвера, но если необходимо, могут создать область контекста в инфраструктурном объекте;
- ◆ драйверы KMDF не создают объектов обратного вызова и поэтому обычно создают область контекста в инфраструктурном объекте.

## Данные контекста объекта UMDF

Драйверы UMDF могут хранить данные контекста для определенного объекта одним из двух способов:

- ◆ в членах данных объекта обратного вызова;
- ◆ в отдельной области контекста, определенной драйвером и выделенной им инфраструктурному объекту.

Наилучшее место для хранения данных контекста зависит от взаимоотношений между инфраструктурным объектом и объектом обратного вызова. Если между инфраструктурным объектом и объектом обратного вызова существует взаимоотношение один к одному, т. е. если данный объект обратного вызова обслуживает только один инфраструктурный объект, наилучшим местом для хранения данных контекста обычно является сам объект обратного вызова. Например, если драйвер создает отдельный объект обратного вызова очереди для каждой очереди, специфичные для конкретной очереди данные можно хранить в элементе данных объекта обратного вызова очереди, вместо того чтобы создавать для них область контекста в инфраструктурном объекте очереди.

Но если объект обратного вызова будет обслуживать несколько инфраструктурных объектов, например, объекты запросов ввода/вывода и объекты файлов, специфичные для объекта данные лучше хранить с каждым индивидуальным инфраструктурным объектом. Драйвер обычно не создает выделенный объект обратного вызова для каждого запроса ввода/вывода, который он получает от инфраструктуры, т. к. это вызвало бы большое количество временных объектов и напрасное расходование памяти. Вместо этого драйвер должен сохранить данные контекста в инфраструктурном объекте.

### Использование объектов обратного вызова и областей контекста

Объекты обратного вызова и области контекста можно использовать несколькими способами.

Для структурных объектов, например устройства или очереди, лучше иметь настоящий объект обратного вызова для каждого инфраструктурного объекта. На случай, если устройству понадобится обратиться к определенной очереди, между объектами обратного вызова можно применять указатели.

Для таких скоротечных событий, как запрос, может быть более разумным реализовать интерфейсы обратного вызова запроса на уже существующем объекте обратного вызова, например, объекте обратного вызова очереди или устройства. Таким образом, не будет необходимости выделять объект обратного вызова для каждого прибывающего запроса.

Если вам нужно сохранить небольшой объем данных контекста для каждого запроса, но вы не хотите создавать новый класс COM для этого, можно использовать комбинацию обратного вызова и области контекста. Создайте интерфейсы обратных вызовов на объекте обратного вызова очереди или устройства, как и прежде, но выделите немного памяти для области контекста инфраструктурного объекта запроса. В этой области контекста можно хранить любую информацию, а аспектами COM интерфейсов обратного вызова занимается устройство или очередь. Но только помните, что также необходимо реализовать обратный

вызовов функции `IObjectCleanup::OnCleanup` для инфраструктурного объекта запроса, с тем, чтобы освободить структуру контекста.

Даже если вы решите создать объект обратного вызова запроса, использование области контекста может все еще оказаться полезным. Скажем, вы выделяете объект обратного вызова, а потом направляете запрос в очередь с ручной обработкой для обработки позже. При извлечении запроса из очереди с ручной обработкой, вам, наверное, будет нужно найти этот объект обратного вызова. Эту задачу можно с легкостью выполнить, сохранив значение указателя на объект обратного вызова в области контекста. В данном случае, область контекста и обратный вызов одинаковы, так что вам не обязательно будет нужно выполнять обратный вызов метода `OnCleanup`; область контекста будет очищена обычным механизмом COM подсчета клиентов. (*Питер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Информация контекста UMDF в членах данных объекта обратного вызова

Чтобы сохранить данные контекста, специфичные для объекта обратного вызова, созданного драйвером, драйвер UMDF может просто объявить элементы данных класса объекта. Таким образом, объект обратного вызова предоставляет собственную память для хранения контекста. Например, в листинге 5.5 показан фрагмент заголовочного файла Device.h образца драйвера Fx2\_Driver, в котором объявляются закрытые элементы данных для объекта обратного вызова устройства.

### Листинг 5.5. Хранение данных контекста в объекте обратного вызова

```
class CMyDevice :  
public CUnknown,  
public IPnPCallbackHardware,  
public IPnPCallback,  
public IRequestCallbackRequestCompletion  
{  
    // Приватные элементы данных  
private:  
    IWDFDevice *          m_FxDevice;  
    PCMyReadWriteQueue    m_ReadWriteQueue;  
    PCMyControlQueue     m_ControlQueue;  
    IWDFUsbTargetDevice * m_pIUsbTargetDevice;  
    IWDFUsbInterface *   m_pIUsbInterface;  
    IWDFUsbTargetPipe *  m_pIUsbInputPipe;  
    IWDFUsbTargetPipe *  m_pIUsbOutputPipe;  
    IWDFUsbTargetPipe *  m_pIUsbInterruptPipe;  
    UCHAR                m_Speed;  
    SWITCH_STATE         m_SwitchState;  
    HRESULT              m_InterruptReadProblem;  
    SWITCH_STATE         m_SwitchStateBuffer;  
    IWDFIOQueue *        m_SwitchChangeQueue;  
}
```

В закрытых элементах данных хранится информация, используемая драйвером с объектом обратного вызова устройства. Это такая информация, как указатель на интерфейс `IWDFDevice` инфраструктуры, указатели на очереди ввода/вывода устройства, а также указатели на интерфейсы инфраструктуры для получателя ввода/вывода USB, интерфейса USB, объектов канала USB и разнообразные переменные состояния.

## Область контекста UMDF для объекта инфраструктуры

Драйвер UMDF может соотнести специфичные для объекта данные с инфраструктурным объектом, таким как, например, объект запроса ввода/вывода или файла. Для этого он создает область контекста и ассоциирует ее с данным объектом. Драйвер может назначить область контекста объекту в любое время, но обычно это делается сразу же после создания объекта. В UMDF инфраструктурный объект может иметь только одну область контекста.

Область контекста — это просто выделенная драйвером область памяти. Драйвер объявляет область контекста, создавая структуру, в которой можно хранить требуемые данные. Чтобы назначить область контекста, драйвер вызывает метод `IWDFObject::AssignContext` объекта, передавая в параметрах указатель на интерфейс `IObjectCleanup` и указатель на область контекста.

Интерфейс `IObjectCleanup` поддерживает метод `OnCleanup`, который освобождает область контекста и любые дополнительные ссылки на объект, полученные драйвером. Инфраструктура вызывает метод `OnCleanup` непосредственно перед тем, как она удалит объект.

В листинге 5.6 показан пример создания драйвером объекта типа `Context`, определенного драйвером, и назначение этого объекта в качестве области контекста для объекта файла. Данный пример взят из драйвера `WpdHelloWorldDriver` в файле `Queue.cpp`.

### Листинг 5.6. Создание области контекста в объекте UMDF

```
Context* pClientContext = new Context();
if(pClientContext != NULL) {
    hr = pFileObject->AssignContext(this, (void*)pClientContext);
    if(FAILED(hr)) {
        pClientContext->Release();
        pClientContext = NULL;
    }
}
```

Область контекста не является по умолчанию объектом COM, хотя драйвер может использовать объект COM, как сделано в предыдущем примере. Если драйвер использует объект COM, забота о подсчете ссылок на объект ложится на него.

В листинге 5.6 драйвер создает новый объект контекста и назначает его объекту файла, вызывая метод `AssignContext`. Драйвер передает указатель на созданный драйвером интерфейс `IObjectCleanup` для области контекста и указатель на сам контекст. В случае неуспешного завершения метода `AssignContext` драйвер освобождает свою ссылку на область контекста и присваивает указателю на контекст значение `NULL`. В листинге 5.7 показан исходный код метода `OnCleanup` для области контекста.

### Листинг 5.7. Метод `OnCleanup` для области контекста UMDF

```
STDMETHODIMP_(void) CQueue::OnCleanup(
    IWDFObject* pWdfObject
)
{
    HRESULT hr      = S_OK;
    Context* pClientContext = NULL;
    hr = pWdfObject->RetrieveContext((void**)&pClientContext);
```

```
if((hr == S_OK) && (pClientContext != NULL))
{
    pClientContext->Release();
    pClientContext = NULL;
}
}
```

Метод `OnCleanup` вызывает метод `IWDFObject::RetrieveContext`, чтобы получить указатель на область контекста. В данном примере область контекста является объектом СОМ, поэтому драйвер освобождает свою ссылку на область контекста, а потом присваивает указателю значение `NULL`.

## Область контекста объекта KMDF

Объекты KMDF могут иметь несколько областей контекста объекта. Инфраструктура выделяет области контекста из нестраничного пула и обнуляет их при инициализации. Область контекста считается частью объекта.

Для объектов устройств и для других объектов, созданных драйвером, драйвер обычно назначает область контекста при создании им объекта. Для объекта устройства, область контекста объекта является эквивалентом расширения устройства в WDM. В действительности, поле `DeviceExtension` структуры `DEVICE_OBJECT` WDM указывает на первую область контекста, назначенную объекту устройства WDF.

Драйвер назначает область контекста объекту, созданному инфраструктурой, после его создания, или области контекста объекту, созданному драйвером, вызывая метод `WdfObjectAllocateContext`. В качестве параметра драйвер передает дескриптор объекта, которому нужно назначить область контекста. Этот дескриптор должен представлять действительный объект. Если объект находится в процессе удаления, вызов завершается неудачей. Если объект имеет больше чем одну область контекста, все они должны быть разных типов.

Драйвер определяет тип и размер каждой области контекста и записывает эту информацию в структуре атрибутов объекта. Структура атрибутов является входным параметром метода создания объекта или метода `WdfObjectAllocateContext`. Инфраструктура предоставляет макросы для сопоставления типа и имени с областью контекста и создания именованной функции доступа, которая возвращает указатель на область контекста. Драйвер должен использовать метод доступа, чтобы получить указатель на область контекста, т. к. область контекста является частью непрозрачного для драйвера объекта. Каждая область контекста имеет свой собственный метод доступа.

### Организация области контекста

Если вы знакомы с моделью WDM, организация области контекста в WDF может показаться вам необоснованно сложной. Но применение областей контекста позволяет гибкость в присоединении информации к запросам ввода/вывода при их прохождении через драйвер и позволяет разным библиотекам иметь свой собственный контекст для объекта. Например, библиотека IEEE 1394 может отслеживать устройство `WDFDEVICE` одновременно с отслеживанием этого же устройства драйвером функции устройства, но применяя отдельный контекст в каждом случае.

Только объект, создавший область контекста, может иметь доступ к ней. Внутри драйвера область контекста позволяет абстрагировать и инкапсулировать данные. Если драйвер использует запрос для выполнения нескольких разных задач, объект запроса может иметь отдельную область контекста для каждой задачи. Функции, связанные с выполнением определенной задачи, могут иметь свои собственные контексты и не требуют никакой ин-

формации о существовании или содержимом любых других контекстов. (*Даун Холэн (Down Holan), команда разработчиков Windows Driver Foundation, Microsoft.*)

Когда инфраструктура KMDF удаляет объект, она также удаляет его области контекста. Драйвер не может динамически удалить область контекста. Области контекста продолжают существовать до тех пор, пока объект не удален.

Чтобы установить область контекста, нужно выполнить следующие операции:

1. Объявить тип области контекста.
2. Инициализировать структуру атрибутов объекта информацией об области контекста.
3. Назначить область контекста объекту.

## Объявления типа области контекста KMDF

Тип области контекста объявляется посредством макроса `WDF_DECLARE_CONTEXT_TYPE_WITH_NAME` в заголовочном файле.

Макросы для объявления типа контекста ассоциируют тип с областью контекста и создают именованный метод доступа, который возвращает указатель на область контекста.

Макрос `WDF_DECLARE_CONTEXT_TYPE_WITH_NAME` назначает указанное драйвером имя методу доступа.

Например, фрагмент кода в листинге 5.8 (взятый из файла Nonprp.h в образце драйвера Nonprp) определяет тип контекста для объекта запроса ввода/вывода.

### Листинг 5.8. Объявление типа контекста для объекта KMDF

```
typedef struct _REQUEST_CONTEXT {
    WDFMEMORY InputMemoryBuffer;
    WDFMEMORY OutputMemoryBuffer;
} REQUEST_CONTEXT, *PREQUEST_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(REQUEST_CONTEXT, GetRequestContext)
```

В листинге 5.8 объявляется контекст типа `REQUEST_CONTEXT` и присваивается имя `GetRequestContext` его методу доступа.

## Инициализация полей контекста в структуре атрибутов объекта

Драйвер записывает тип области контекста в структуру атрибутов объекта с помощью макроса `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE` или макроса `WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE`.

Макрос `WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE` записывает информацию об области контекста в ранее инициализированной структуре атрибутов, которую драйвер предоставляет в дальнейшем при создании объекта.

Макрос `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE` совмещает действия макросов `WDF_OBJECT_ATTRIBUTES_INIT` и `WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE`.

То есть, кроме информации о контексте, он инициализирует структуру атрибутов установками для других атрибутов.

В листинге 5.9 показан пример инициализации структуры атрибутов информацией об области контекста, определенной в шаге 1.

**Листинг 5.9. Инициализация структуры атрибутов информацией об области контекста**

```
WDF_OBJECT_ATTRIBUTES attributes;  
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, REQUEST_CONTEXT);
```

В предыдущем листинге структура атрибутов инициализируется информацией о типе области контекста REQUEST\_CONTEXT.

**Назначение области контекста KMDF объекту**

Драйвер KMDF может ассоциировать область контекста с новым или уже существующим объектом. Чтобы ассоциировать область контекста с новым объектом, драйвер передает инициализированную структуру атрибутов при вызове метода создания объекта.

Чтобы ассоциировать область контекста с уже существующим объектом, драйвер вызывает метод WdfObjectAllocateContext и передает в нем три параметра: дескриптор объекта, указатель на инициализированную структуру атрибутов и адрес, по которому метод возвращает указатель на выделенную область контекста.

В листинге 5.10 показан пример выделения области контекста, которая была установлена в предыдущих двух листингах, и назначения ее объекту запроса Request.

**Листинг 5.10. Назначение области контекста KMDF существующему объекту**

```
status = WdfObjectAllocateContext(Request, &attributes, &reqContext);
```

Когда драйвер удаляет объект запроса, инфраструктура также удаляет и область контекста.

# ГЛАВА 6

## Структура драйвера и его инициализация

В этой главе описываются базовые возможности, которые должен иметь любой драйвер, и приводится объяснение, каким образом драйвер создает и инициализирует два своих самых важных объекта: объект драйвера и объект устройства.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы WDK</b>	
Simple Toaster	%wdk%\src\kmdf\toaster\func\simple
Fx2_Driver	%wdk%\src\umdf\usb\fx2_driver
Osrusbf2	%wdk%\src\kmdf\osrusbf2
<b>Документация WDK</b>	
Device Interface Classes <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=81577">http://go.microsoft.com/fwlink/?LinkId=81577</a>
Using Device Interfaces <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=81578">http://go.microsoft.com/fwlink/?LinkId=81578</a>
Securing Device Objects <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80624">http://go.microsoft.com/fwlink/?LinkId=80624</a>
Creating Secure Device Installations <sup>4</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80625">http://go.microsoft.com/fwlink/?LinkId=80625</a>

## Обязательные компоненты драйвера

Независимо от режима исполнения, каждый драйвер должен реализовывать определенные функции и использовать определенные объекты. А именно:

- ◆ точку входа, т. е. функцию, которая вызывается операционной системой при загрузке драйвера;
- ◆ объект драйвера, представляющий драйвер;

<sup>1</sup> Классы интерфейса устройства. — *Пер.*

<sup>2</sup> Использование интерфейсов устройства. — *Пер.*

<sup>3</sup> Организация защиты объектов устройства. — *Пер.*

<sup>4</sup> Создание безопасных установочных пакетов драйверов. — *Пер.*

- ◆ один или несколько объектов устройств, представляющие устройства, контролируемые драйвером;
- ◆ дополнительные объекты, используемые драйвером с объектами устройств для управления устройством и потоком запросов ввода/вывода к устройству;
- ◆ обратные вызовы для обработки событий, находящихся в сфере ответственности драйвера.

Каждый драйвер имеет объект драйвера, который представляет драйвер в инфраструктуре. Драйвер предоставляет инфраструктуре информацию об обратных вызовах, которые он поддерживает для объекта драйвера.

Когда система перечисляет управляемое драйвером устройство, драйвер создает объект устройства для представления данного устройства и предоставляет информацию об обратных вызовах по событию для объектов устройства. Драйвер также создает очереди ввода/вывода для обработки входящих запросов для объекта устройства. Многие драйверы также создают дополнительные вспомогательные объекты, включая объекты исполнителя ввода/вывода, которые представляют исполнителей запросов ввода/вывода драйвера. После создания драйвером объекта драйвера, объектов устройства и очередей и вспомогательных объектов рабочая структура драйвера (т. е. его внутренняя инфраструктура) готова и инициализация практически завершена.

Хотя как UMDF-, так и KMDF-драйверы требуют объектов и структур одинакового типа, во многих отношениях они реализуются по-разному для каждого типа драйвера, что и будет показано далее в этой главе.

## Структура драйверов UMDF и требования к ним

Все драйверы UMDF должны выполнять следующие задачи:

- ◆ реализовать функцию `DllMain` в качестве точки входа драйвера;
- ◆ реализовать и экспорттировать по имени функцию `DllGetClassObject`;
- ◆ реализовать интерфейс `IClassFactory` для создания объекта драйвера;
- ◆ реализовать объект обратного вызова драйвера, предоставляющий интерфейс `IDriverEntry`;
- ◆ реализовать объект обратного вызова драйвера, предоставляющий интерфейсы обратного вызова для объекта устройства.

Кроме этого, все драйверы UMDF создают одну или несколько очередей ввода/вывода. Каждая очередь имеет соответствующий объект обратного вызова, который предоставляет интерфейсы обратного вызова для событий ввода/вывода, обрабатываемых драйвером. Драйверы также могут создавать дополнительные объекты для предоставления поддержки для любых других задач.

Дополнительная информация о функциях `DllMain`, `DllGetClassObject` и `IClassFactory` приводится в главе 18.

Интерфейс содержит методы для инициализации и deinициализации драйвера и для создания объекта устройства при добавлении устройства в систему. Инфраструктура вызывает эти методы при загрузке или выгрузке драйвера, а также при перечислении PnP-менеджером одного из устройств драйвера.

На рис. 6.1 показан поток управления при загрузке, инициализации и работе драйвера UMDF.

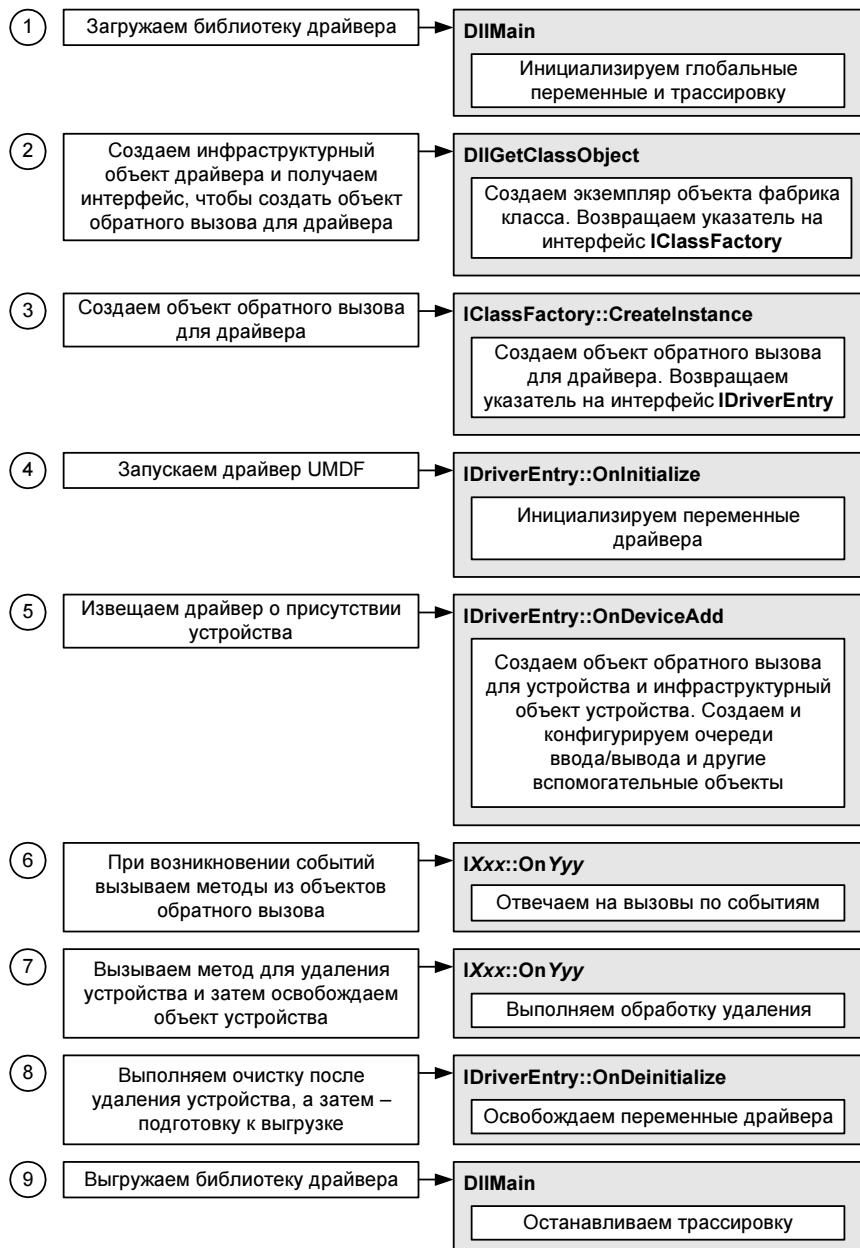


Рис. 6.1. Поток управления для драйверов UMDF

При своем запуске Windows загружает менеджер драйверов, после чего выполняется такая последовательность действий:

1. Менеджер драйверов создает новый хост-процесс драйвера и инфраструктурный объект драйвера, после чего вызывает системный загрузчик библиотек DLL, который загружает DLL драйвера, вызывая функцию точки входа `DllMain`. Функция `DllMain` выполняет определенные глобальные инициализации драйвера, например, запускает трассировку. Сис-

тема накладывает некоторые ограничения на тип операций, которые функция `DllMain` может выполнять. Наиболее важным из них является запрет на любые действия, в результате которых загружается другая библиотека DLL. Вместо этого, большая часть инициализации драйвера должна выполняться в методе `OnInitialize`.

2. Инфраструктура вызывает функцию `DllGetClassObject` драйвера, чтобы получить указатель на интерфейс `IClassFactory`, который может создать объект обратного вызова драйвера в драйвере. Функция `DllGetClassObject` возвращает указатель на интерфейс `IClassFactory` драйвера.
3. Инфраструктура вызывает метод `IClassFactory::CreateInstance`, чтобы создать экземпляр объекта обратного вызова драйвера.

Объект обратного вызова драйвера предоставляет интерфейс `IDriverEntry`, который содержит методы для инициализации драйвера, информирования его о том, что одно из его устройств было перечислено, и подготовки драйвера к выгрузке.

4. Инфраструктура вызывает метод `IDriverEntry::OnInitialize`, чтобы инициализировать драйвер. Метод `OnInitialize` инициализирует глобальные данные драйвера и выполняет любые другие задачи, которые не могут быть выполнены в функции `DllMain`.
5. При перечислении устройства драйвера инфраструктура вызывает метод `IDriverEntry::OnDeviceAdd`.

Этот метод выполняет все необходимое конфигурирование, создает объект обратного вызова устройства для представления устройства, создает все необходимые интерфейсы устройства, создает и конфигурирует очереди, в которые инфраструктура будет помещать запросы ввода/вывода, направленные драйверу.

### Примечание

Интерфейс устройства описывает набор функциональных возможностей, предоставляемых всем устройством, которые драйвер предоставляет приложениям или другим компонентам системы. А интерфейс COM представляет собой группу родственных функций, которые составляют функциональные возможности конкретного внутреннего объекта.

6. При возникновении событий инфраструктура активирует методы на объектах обратного вызова для их обработки. Инфраструктура получает указатели на интерфейсы обратного вызова, вызывая метод `QueryInterface` на объектах обратного вызова.
7. При удалении устройства инфраструктура вызывает обратные вызовы релевантного драйвера, а потом освобождает объекты обратного вызова устройства.
8. Инфраструктура вызывает метод `IDriverEntry::OnDeinitialize`, чтобы очистить ресурсы после удаления устройства, а потом освобождает объект драйвера.
9. Инфраструктура вызывает функцию `DllMain`, выгружает DLL и удаляет хост-процесс драйвера.

Дополнительную информацию о функции `DllMain` можно найти в разделе **DllMain** в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=80069>.

### Инициализация драйверов

В настоящее время UMDF загружает в хост-процесс драйверы только для одного стека устройства. Поэтому функция `DllGetClassObject` и методы `IDriverEntry::Xxx` вызываются только один раз для каждого устройства. Другое устройство, использующее тот же драйвер, загружается в другой хост-процесс, и эти функции инициализации вызываются опять.

По этой причине метод `IDriverEntry::OnInitialize` может казаться излишним. Драйвер загружается только один раз, так почему бы не выполнять всю инициализацию драйвера в функции `DllMain` или в конструкторе класса драйвера?

Система накладывает определенные ограничения на операции, которые DLL может выполнять, не вызывая проблем, в функции `DllMain`, поэтому лучше выполнять как можно меньше операций в этой функции, с тем, чтобы избежать риска заблокировать загрузчик. Для создания чистого объекта обратного вызова драйвера пользуйтесь конструктором класса драйвера, методом `IDriverEntry::OnInitialize` — для всеобщей инициализации драйвера, а методом `OnDeinitialize` — для его деинициализации. (Питер Вилант (*Peter Wieland*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Структура драйверов KMDF и требования к ним

Драйвер KMDF состоит из функции `DriverEntry`, которая идентифицирует драйвер как драйвер KMDF, набора функций обратного вызова, вызываемых инфраструктурой для обработки драйвером событий, связанных с его устройством, а также других специфичных для драйвера служебных функций. Все драйверы KMDF должны иметь следующие компоненты.

- ◆ Функцию `DriverEntry`, которая является главной точкой входа драйвера и создает объект драйвера WDF.
- ◆ Функцию обратного вызова по событию `EvtDriverDeviceAdd`, которая вызывается, когда PnP-менеджер обнаруживает одно из устройств драйвера.

Эта функция обратного вызова создает и инициализирует объект устройства, объекты очередей и другие вспомогательные объекты, необходимые для работы драйвера.

Функция `EvtDriverDeviceAdd` не требуется для драйверов устройств, не поддерживающих Plug and Play.

- ◆ Одну или несколько функций обратного вызова `EvtXxx`, для обработки событий, возникающих во время работы драйвера.

Драйвер KMDF с минимальными возможностями для простого устройства может иметь только эти и никаких больше функций. Инфраструктура реализует стандартное управление электропитанием и операции Plug and Play, поэтому для драйверов, которые не работают с физическим оборудованием, не требуется много кода для выполнения этих задач. Если драйвер может обойтись стандартными функциональными возможностями, для него не нужно разрабатывать код для выполнения многих общих задач. Но чем больше специфичных для устройства возможностей должен поддерживать драйвер и чем больше функциональных возможностей он предоставляет, тем больше необходимо разрабатывать для него код.

Файл `Toaster.c` из примера драйвера Simple Toaster служит отличным примером, насколько простым может быть загружаемый, рабочий драйвер KMDF. За исключением кода заголовочных файлов в этом файле содержится весь исходный код драйвера Simple Toaster — менее чем 400 строчек. Но этого кода достаточно, чтобы создать интерфейс устройства и очередь ввода/вывода и обрабатывать запросы на чтение, запись и IOCTL.

Для драйверов KMDF для устройств Plug and Play не требуется функция выгрузки драйвера, т. к. инфраструктура предоставляет эту функцию по умолчанию. Но если драйвер создает или выделяет глобальные для драйвера ресурсы в функции `DriverEntry` и использует их до тех пор, пока он не выгрузится, драйвер должен содержать функцию `EvtDriverUnload` для освобождения этих ресурсов.

Не-Pnp-драйвер может дополнительно зарегистрировать функцию обратного вызова `EvtDriverUnload`. Если драйвер не предоставит обратного вызова, то его нельзя будет выгрузить.

На рис. 6.2 показан поток управления при загрузке, инициализации и работе драйвера KMDF.

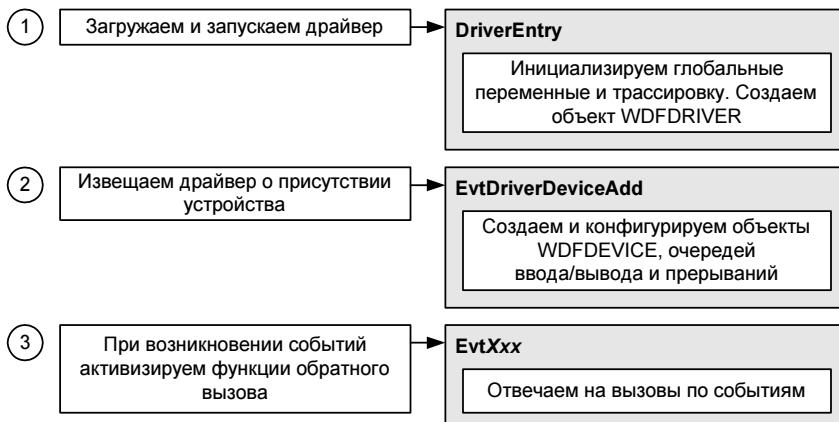


Рис. 6.2. Поток управления для драйверов KMDF

Когда Windows загружает драйвер KMDF, он динамически привязывается к совместимой версии библиотеки времени выполнения KMDF. Образ драйвера содержит информацию о версии KMDF, для которой он был создан:

1. Загрузчик WDF определяет, была ли уже загружена требуемая основная версия библиотеки инфраструктуры.

Если требуемая версия библиотеки не была загружена, то драйвер загружает ее. Если драйверу требуется более новая дополнительная версия библиотеки времени выполнения, чем версия, загруженная в данный момент, загрузка завершается неудачей, о чем делается соответствующая запись в системном журнале событий. В таком случае, последующие действия не выполняются.

Если требуемая версия была загружена, загрузчик добавляет драйвер как клиент сервиса и возвращает релевантную информацию инфраструктуре. Инфраструктура вызывает функцию `DriverEntry` драйвера.

Функция `DriverEntry` инициализирует глобальные для драйвера данные, инициализирует трассировку и вызывает метод `WdfDriverCreate`, чтобы создать инфраструктурный объект драйвера.

2. Инфраструктура вызывает функцию обратного вызова `EvtDriverDeviceAdd`, которую драйвер зарегистрировал при создании объекта драйвера.

Функция обратного вызова `EvtDriverDeviceAdd` создает объект устройства и инициализирует его, создает и конфигурирует очереди ввода/вывода для объекта устройства, создает дополнительные вспомогательные объекты, требуемые для объекта устройства и объектов очередей, регистрирует функции обратного вызова по событию для созданных ею объектов и регистрирует требуемые интерфейсы устройств.

3. Инфраструктура вызывает функции *EvtXXX* драйвера для обработки событий. Если добавляются дополнительные управляемые данным драйвером устройства, функция *EvtDriverDeviceAdd* вызывается для каждого из этих устройств.

Удаление устройства Plug and Play является событием, для которого драйвер может зарегистрировать функцию обратного вызова. При удалении устройства инфраструктура вызывает соответствующую функцию обратного вызова для события удаления, а потом вызывает функции обратного вызова для события очистки ресурсов, которые драйвер мог зарегистрировать.

## Объект драйвера

Каждый драйвер имеет объект драйвера, который поддерживает обратный вызов функции для обработки события добавления устройства. Объект драйвера может хранить данные, глобальные для драйвера, а также может поддерживать функцию обратного вызова для очистки этих данных при выгрузке драйвера.

### Создание объекта обратного вызова для драйвера UMDF

Для драйвера UMDF инфраструктура создает инфраструктурный объект драйвера, а потом вызывает метод *IClassFactory::CreateInstance* драйвера на фабрике класса объекта обратного вызова драйвера, который создает объект обратного вызова драйвера. Объект обратного вызова драйвера создает интерфейс *IDriverEntry*, который поддерживает методы, перечисленные в табл. 6.1.

**Таблица 6.1. Методы интерфейса *IDriverEntry***

Метод	Описание
<i>OnInitialize</i>	Инициализирует глобальные для драйвера данные для объекта обратного вызова драйвера и выполняет любые другие задачи инициализации, которые не могут быть выполненными в функции точки входа <i>DllMain</i> . Этот метод вызывается перед вызовом метода <i>OnDeviceAdd</i>
<i>OnDeviceAdd</i>	Создает и инициализирует объект обратного вызова устройства
<i>OnDeinitialize</i>	Освобождает ресурсы, выделенные методом <i>OnInitialize</i> . Часто этот метод реализуется только с минимальными возможностями

Код для реализации метода *OnDeviceAdd* содержится в файле *Driver.cpp* образца драйвера *Fx2\_Driver*. Методы *OnInitialize* и *OnDeinitialize* реализуются лишь с минимальными возможностями. Их определения находятся в файле *Driver.h*.

В драйвере *Fx2\_Driver* и других образцах драйверов UMDF метод *IClassFactory::CreateInstance* вызывает метод *CMyDriver::CreateInstance*, который создает и инициализирует объект обратного вызова драйвера. Хотя метод *IClassFactory::CreateInstance* мог бы создать объект класса драйвера прямым образом, применение метода, определенного для класса драйвера, позволяет оставить исходный код в файле *Comsup.cpp* обобщенным, что делает его доступным для использования любым драйвером. Кроме этого, исходный код всех имеющих отношение к классу драйвера методов можно разместить в одном и том же файле.

Метод `CMyDriver::CreateInstance` определен в исходном файле `Driver.cpp`. Как можно видеть в листинге 6.1, его исходный код прост и ясен.

#### Листинг 6.1. Создание объекта обратного вызова драйвера UMDF

```
HRESULT CMyDriver::CreateInstance(
    __out PCMyDriver *Driver
)
{
    PCMyDriver driver;
    HRESULT hr;
    // Выделяем объект обратного вызова.
    driver = new CMyDrive();
    if (NULL == driver) {
        return E_OUTOFMEMORY;
    }
    // Инициализируем объект обратного вызова.
    hr = driver->Initialize();
    if (SUCCEEDED(hr)) {
        // Возвращаем указатель на только что созданный и еще
        // неинициализированный объект.
        *Driver = driver;
    }
    else {
        // Освобождаем ссылку на объект драйвера.
        driver->Release();
    }
    return hr;
}
```

Метод `CMyDriver::CreateInstance`, чье определение показано в предыдущем листинге, вызывается методом `IClassFactory::CreateInstance`. Метод `IClassFactory::CreateInstance` создает экземпляр объекта обратного вызова драйвера с помощью оператора `new`, после чего инициализирует созданный объект, вызывая метод `Initialize`. Объект `Fx2_Driver` не требует инициализации, поэтому метод `Intialize` реализован в виде маркера, и его определение не показано здесь. Метод `CMyDriver::CreateInstance` возвращает указатель на созданный объект обратного вызова драйвера и освобождает свою ссылку на этот объект перед передачей управления.

После возвращения управления методом `CMyDriver::CreateInstance` методу `IClassFactory::CreateInstance` последний вызывает метод `QueryInterface`, чтобы получить указатель на интерфейс `IDriverEntry` объекта обратного вызова драйвера, и возвращает этот указатель инфраструктуре. Код для этой операции здесь не показан.

#### О шаблоне программирования образцов UMDF

Когда я впервые предложил шаблон программирования образцов для UMDF, коллеги думали, что он был несколько "тяжеловат". Зачем вызывать метод `Initialize` или `Configure`, если они ничего не делают?

Но моей целью было, чтобы образец можно было использовать в качестве шаблона, с которого начинать разработку нового драйвера. Поэтому я решил, что шаблоны для создания объектов обратного вызова и для связки объектов инфраструктуры вместе должны быть однообразными для всех образцов.

Объекты создаются в следующем порядке:

1. Функция, создающая объект, вызывает стандартный метод фабрики `CMuXxx::CreateInstance`. Таким образом, осуществляется выделение памяти, инициализация конструктора и выделение любых других необходимых данных и объектов. Но в основном это гарантирует, что программист не забудет вызвать метод `Initialize`.
2. Конструктор инициализирует все поля объекта известными значениями, но не выделяет память. Этот конструктор приватный, поэтому вызывающие процедуры должны использовать метод `CreateInstance`.
3. С помощью метода `Initialize` выполняется более глубокая инициализация, включая установку свойств инфраструктуры, создание объекта-партнера `IWDFXXX`, установка его в качестве объекта обратного вызова и т. д.
4. Метод `Configure`, который обычно вызывается после методов `CreateInstance` и `Initialize`, конфигурирует дочерние объекты. Например, метод `CMDevice::Configure` в образце драйвера `Fx2_driver` создает все очереди устройства. Метод `Configure` является отдельным от метода `Initialize`, чтобы код, создающий объект, мог выполнять любые необходимые операции перед созданием дочерних объектов.

(Питер Виленд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.)

## Создание объекта драйвера KMDF

Драйвер KMDF создает свой объект драйвера в функции `DriverEntry`, первой функцией, вызываемой при загрузке драйвера. Функция `DriverEntry` вызывается только один раз. Она выполняет следующие действия:

- ◆ создает объект драйвера (т. е. `WDFDRIVER`), который представляет загруженный в память экземпляр драйвера;
- ◆ в сущности, создание этого объекта и "регистрирует" драйвер в инфраструктуре;
- ◆ регистрирует функцию обратного вызова `EvtDriverDeviceAdd` драйвера;
- ◆ инфраструктура вызывает эту функцию, когда PnP-менеджер обнаруживает присутствие устройства;
- ◆ дополнительно инициализирует трассировку событий для драйвера;
- ◆ дополнительно выделяет ресурсы, требуемые глобально для драйвера, в отличие от ресурсов для каждого устройства;
- ◆ если перед выгрузкой драйверу необходимо выполнить определенные операции, регистрирует предназначенную для этого функцию обратного вызова `EvtDriverUnload`.

При успешном создании объекта драйвера и выполнении любых других операций инициализации, требуемых драйвером, функция `DriverEntry` должна возвратить `STATUS_SUCCESS`. При неуспешном завершении выполнения функция `DriverEntry` должна возвратить одно из значений статуса ошибки. В таком случае инфраструктура удаляет объект драйвера (если он был успешно создан), и поэтому не вызывает функцию обратного вызова `EvtDriverUnload`. Исходный код версии функции `DriverEntry` из файла `Driver.c` для образца драйвера `Osrusbfx2` показан в листинге 6.2.

### Листинг 6.2. Пример KMDF-функции `DriverEntry`

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

```

{
    WDF_DRIVER_CONFIG           config;
    NTSTATUS                   status;
    WDF_OBJECT_ATTRIBUTES       attributes;
    // Инициализация структуры конфигурации драйвера.
    WDF_DRIVER_CONFIG_INIT (&config, OsrFxEvtDeviceAdd);
    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    attributes.EvtCleanupCallback = OsrFxEvtDriverContextCleanup;
    // Создание инфраструктурного объекта драйвера.
    status = WdfDriverCreate (DriverObject, RegistryPath,
                            &attributes, // Атрибуты объекта драйвера.
                            &config,      // Конфигурационная информация драйвера.
                            WDF_NO_HANDLE // hDriver
    );
    if (!NT_SUCCESS(status)) {
        return status;
    }
    // Инициализация трассировки WPP.
    WPP_INIT_TRACINC(DriverObject, RegistryPath);
    TraceEvents(TRACE_LEVEL_INFORMATION, DBC_INIT,
                "OSRUSBFX2 Driver Sample – Driver Framework Edition.\n");
    return status;
}

```

Как можно видеть, функция `DriverEntry` имеет два параметра: указатель на лежащий в основе объект драйвера WDM и указатель на путь реестра. Если вы знакомы с драйверами WDM, то, наверное, заметили, что это такие же параметры, как и для функции `DriverEntry` WDM. В сущности, до тех пор, пока функция `DriverEntry` не вызовет метод `WdfDriverCreate`, драйвер фактически является драйвером WDM.

Первым делом функция `DriverEntry` инициализирует структуру конфигурации объекта драйвера указателем на обратный вызов `EvtDriverDeviceAdd` драйвера. Для этого она вызывает функцию `WDF_DRIVER_CONFIG_INIT`.

Далее, функция `DriverEntry` образца драйвера регистрирует обратный вызов функции `EvtCleanupCallback`, устанавливая соответствующее поле в структуре `WDF_OBJECT_ATTRIBUTES`. Инфраструктура выполняет обратный вызов этой функции непосредственно перед удалением объекта драйвера. Функция `EvtCleanupCallback` должна выполнить очистку объекта драйвера, например, освободить ресурсы или, в данном случае, остановить трассировку.

После инициализации структур конфигурации и атрибутов драйвер вызывает метод `WdfDriverCreate`, чтобы создать инфраструктурный объект драйвера. Метод `WdfDriverCreate` принимает следующие параметры:

- ◆ указатель на объект драйвера WDM, который был передан функции `DriverEntry`;
- ◆ указатель на путь реестра, который был передан функции `DriverEntry`;
- ◆ указатель на структуру атрибутов;
- ◆ указатель на структуру конфигурации;
- ◆ адрес для дескриптора созданного объекта `WDFDRIVER`. Это необязательный параметр и может быть `WDF_NO_HANDLE` (определен как `NULL`), если драйвер не нуждается в этом дескрипторе.

Большинство драйверов не сохраняет дескриптор объекта драйвера, т. к. он редко используется, и драйвер всегда может получить его с помощью метода `WdfGetDriver`.

Если методу `WdfDriverCreate` не удастся создать объект драйвера, функция `DriverEntry` прекращает исполнение и возвращает инфраструктуре значение ошибки.

В случае удачного исполнения метода `WdfDriverCreate`, драйвер инициализирует трассировку, вызывая макрос `WPP_INIT_TRACING`, и записывает трассировочное сообщение в журнал.

Тема трассировки рассматривается в *главе 11*.

Драйвер инициализирует трассировку после создания объекта драйвера для того, чтобы вызывать макрос `WPP_CLEANUP` для прекращения трассировки только один раз, в функции `EvtCleanupCallback` для объекта драйвера. Если драйвер выполняет другие глобальные для драйвера задачи инициализации, код для них должен выполняться после инициализации драйвером трассировки, с тем чтобы драйвер мог записывать в журнал возможные ошибки.

Наконец функция `DriverEntry` возвращает значение `NTSTATUS`. Если после успешного создания объекта драйвера методом `WdfDriverCreate` исполнение функция `DriverEntry` завершается неудачно, инфраструктура удаляет объект драйвера и вызывает его функцию `EvtCleanupCallback`.

### Совет

В *главе 24* рассматривается вопрос, каким образом вставлять примечания в функции обратного вызова драйвера, чтобы SDV мог анализировать их на удовлетворение правилам KMDF. Согласно требованиям этих правил, драйвер KMDF должен вызывать метод `WdfDriverCreate` из своей функции `DriverEntry`.

## Объекты устройств

Когда PnP-менеджер обнаруживает устройство, управляемое драйвером, он извещает об этом инфраструктуру. Инфраструктура, в свою очередь, вызывает драйвер, чтобы он мог создать и инициализировать структуры данных, требуемые для управления устройством. Наиболее важным из этих структур является объект устройства.

Объект устройства представляет роль драйвера в управлении устройством. Он содержит информацию о статусе и является исполнителем драйвера для запросов ввода/вывода, направленных данному устройству. Windows также направляет запросы объекту устройства, а не драйверу.

Объект устройства содержит специфическую для устройства информацию, которую драйвер использует при обработке запросов ввода/вывода и управлении устройством. Объект устройства также поддерживает функции обратного вызова, реализуемые драйвером для реагирования на события устройства.

## Типы объектов устройств

Функциональный драйвер, драйвер шины, а также любые драйверы фильтров устройства реагируют на многие одинаковые события и обрабатывают многие одинаковые запросы. Но это не означает, что любое событие или запрос может быть обработано любым типом драйвера. Например, драйверы шины реагируют на события, не затрагивающие функциональных драйверов, и в некоторых случаях требуют доступа к другим свойствам устройства и прочей

информации об устройстве. Поэтому WDF определяет типы объектов устройств для каждого типа драйвера.

Это следующие типы объектов устройств.

◆ **Объект устройства фильтра (filter device object, FiDO).**

Драйвер фильтра создает объект FiDO для каждого из своих устройств. Объекты устройств фильтра "фильтруют", другими словами — модифицируют, один или несколько типов запросов ввода/вывода, направленных устройству.

◆ **Объект функционального устройства (functional device object, FDO).**

Функциональный драйвер создает объект FDO для каждого из своих устройств. Этот объект FDO представляет основной драйвер устройства.

◆ **Объект физического устройства (physical device object, PDO).**

Драйвер шины создает объект PDO для каждого подключенного к шине устройства. Объект PDO представляет устройство по отношению к его шине.

Драйверы UMDF не могут создавать объекты PDO.

◆ **Объект устройства управления (control device object, CDO).**

Любой драйвер KMDF может создать объект CDO, который представляет наследуемое устройство, не поддерживающее Plug and Play, или интерфейс управления, посредством которого драйвер Plug and Play получает так называемые запросы ввода/вывода "боковой полосы". Объекты CDO не являются частью стека устройств Plug and Play.

Драйверы UMDF не могут создавать объекты CDO.

Хотя в этой книге основное внимание уделяется созданию драйверов функций и фильтров, вы должны быть знакомы с драйверами всех типов и со всеми объектами устройств. Краткое обозрение типов драйверов и соответствующих объектов приводится в следующем разделе.

## Драйверы фильтра и объекты устройств фильтра

Драйверы фильтра сами обычно не выполняют ввод/вывод для устройства, а модифицируют или записывают запрос, который потом выполняется другим драйвером. Например, функция шифрования и дешифрования данных, специфичных для устройства, обычно реализуется в виде драйвера фильтра. Драйвер фильтра получает один или несколько типов запросов ввода/вывода, направленных его устройству, выполняет на основе запроса некоторое действие, а потом обычно передает запрос следующему драйверу в стеке.

Драйвер фильтра добавляет объект FiDO в стек устройств. Драйвер извещает инфраструктуру о том, что он является драйвером фильтра при добавлении устройства в систему, чтобы инфраструктура могла установить соответствующие настройки по умолчанию.

Большинство драйверов фильтра не "заинтересованы" во всех без исключения запросах. Например, драйвер фильтра может фильтровать только запросы на создание или чтение записей. Драйвер фильтра организует очереди для запросов тех типов, которые он фильтрует. Инфраструктура посыпает драйверу только эти типы запросов, а другие типы запросов передает дальше вниз по стеку устройств. Драйвер фильтра никогда не получает эти запросы и поэтому для него не требуется код для их анализа и передачи вниз по стеку.

Некоторые типы устройств могут также иметь драйверы фильтра шины, которые исполняют сложные низкоуровневые операции шины. Драйверы фильтра шины встречаются редко, и WDF не поддерживает их разработку.

## **Функциональные драйверы и объекты функциональных устройств**

Функциональный драйвер является основным драйвером устройства. Он взаимодействует со своим устройством для выполнения ввода/вывода и обычно управляет политикой энергопотребления устройства. Например, он определяет, когда устройство пристаивает и его энергопотребление может быть уменьшено для экономии электроэнергии. В стеке устройств Plug and Play функциональный драйвер предоставляет объект FDO. По умолчанию при создании драйвером объекта устройства обе инфраструктуры создают объект FDO, если только драйвер не указывает иное.

К функциональным драйверам ядра часто предъявляются дополнительные требования, которые не предъявляются к функциональным драйверам пользовательского режима. Например, функциональный драйвер, управляющий устройством, поддерживающим сигналы пробуждения, должен реализовать функции обратного вызова для разрешения и запрещения таких сигналов. Интерфейсы DDI `WdfFdoInitXXX` и `WdfFdoXXX` определяют набор методов, событий и свойств, применимых к объекту FDO драйвера KMDF во время инициализации и работы.

С помощью этих интерфейсов FDO драйвер может выполнять следующие операции:

- ◆ регистрировать обратные вызовы функций для событий, связанных с выделением ресурсов его устройству;
- ◆ получать значения свойств своего физического устройства.

Большинство образцов драйверов создает объект FDO, за исключением KMDF-драйверов `KbFiltr`, `Toaster Filter` и `Firefly`.

## **Драйверы шины и объекты физических устройств (KMDF)**

Драйвер шины управляет родительским устройством, которое перечисляет одно или несколько дочерних устройств. Таким родительским устройством может быть адаптер PCI, хаб USB или подобное устройство, в которое пользователь может подключать другие устройства. Родительским устройством также может быть многофункциональное устройство, которое перечисляет постоянные дочерние устройства, каждое требующее своего типа драйвера. Примером такого устройства может служить звуковая плата с интегрированным звуковым портом и игровым портом. В этой книге все родительские устройства такого типа называются "шинами", а их драйверы — драйверами шины.

Важным различием между драйвером шины и другими типами драйверов является то, что драйвер шины управляет оборудованием, которое не является концевым устройством узла `devnode`. Таким образом, драйвер шины отвечает за выполнение следующих задач:

- ◆ обработку запросов ввода/вывода, направленных самойшине;
- ◆ перечисление дочерних устройств и предоставление информации об их аппаратных требованиях и статусе.

В стеке устройств Plug and Play драйвер шины обычно создает, по крайней мере, два объекта устройств: объект FDO для выполнения его роли функционального драйвера для самой шины и объект PDO для каждого дочернего устройства, подключенного к этой шине.

Инфраструктура определяет методы, события и свойства, специфичные для объектов PDO, подобно тому, как это делается для объектов FDO. Посредством методов интерфейсов `WdfPdoInitXXX` и `WdfPdoXXX` драйвер может выполнять следующие операции:

- ◆ регистрирует функции обратного вызова для событий, которые извещают об аппаратных требованиях его потомков;

- ◆ регистрирует функции обратного вызова для событий, связанных с блокировкой и извлечением устройства;
- ◆ регистрирует функции обратного вызова для событий, которые выполняют операции на уровне шины, чтобы его дочерние устройства могли инициировать сигнал пробуждения;
- ◆ назначает идентификаторы Plug and Plug, совместимости и экземпляра своим дочерним устройствам;
- ◆ извещает систему о взаимоотношениях между своими дочерними устройствами, чтобы менеджер PnP мог координировать их удаление и извлечение;
- ◆ извещает систему об извлечении или неожиданном удалении устройства;
- ◆ удаляет и обновляет шинный адрес дочернего устройства.

Драйвер KMDF указывает, что он является драйвером шины, вызывая один или несколько методов инициализации объекта PDO, перед тем, как создать свой объект устройства.

**"Сырые" устройства.** В редких случаях драйвер шины может управлять так называемым "сырым" устройством. "Сырое" устройство — это устройство, которое управляется непосредственно драйвером шины и объектом PDO, без объекта FDO. Драйвер шины может указывать, что объект PDO может работать в "сыром" режиме. Это означает, что устройство может запуститься и быть доступным клиентам даже без функционального драйвера. Например, драйверы шины для устройств хранения SCSI, IDE и других типов создают объект PDO для каждого обнаруженного ими нашине устройства, и помечают эти объекты PDO, как способными работать в сырому режиме. Для стандартных устройств, например для приводов жестких дисков и CD-ROM, система загружает функциональный драйвер, который управляет устройством. Но если для устройства не существует функционального драйвера, система запускает такое устройство в "сыром" режиме. Стек устройств хранения поддерживает команды свободного прохода для устройств хранения (storage pass-through command), с помощью которых клиент может издать прямые запросы SCSI сырому устройству. Не все шины поддерживают такие команды свободного прохода, но те, которые поддерживают, обычно помечают свои объекты PDO, как поддерживающие сырой режим.

Если драйвер указывает, что он управляет сырым устройством, инфраструктура полагает, что этот драйвер является менеджером политики энергопотребления данного устройства. Драйвер может изменить эту установку методом `wdfDeviceInitSetPowerPolicyOwnership`, но в таком случае необходимо, чтобы другой драйвер управлял энергопотреблением для данного устройства.

**Модели перечислений.** Инфраструктура поддерживает как статическую, так и динамическую модель перечисления дочерних устройств. При статическом перечислении драйвер обнаруживает и извещает о дочерних устройствах во время инициализации системы; его возможности для предоставления информации о последующих изменениях конфигурации ограничены. Драйвер шины должен применять статическую модель перечисления в случаях, когда статус дочерних устройств меняется не часто. Таким образом, статическое перечисление подходит для большинства многофункциональных устройств.

Динамическая модель перечисления поддерживает драйверы, которые могут обнаруживать и докладывать об изменениях количества и типов устройств, подключенных к шине, при штатном режиме работы системы. Драйверы шины должны применять динамическое перечисление в тех случаях, когда количество или тип устройств, подключенных к родительскому устройству, зависит от конфигурации системы. Некоторые из этих устройств могут быть подключенными к системе постоянно, а некоторые можно физически подключать или от-

ключать во время штатной работы системы. Динамическая модель перечисления поддерживает драйверы для таких устройств, для которых статус дочерних устройств может изменяться в любое время, например контроллеры IEEE 1394.

Большинство подробностей процесса перечисления для драйверов шины выполняется инфраструктурой, включая следующие задачи:

- ◆ информирование системы о дочерних устройствах;
- ◆ координирование сканирования для обнаружения дочерних устройств;
- ◆ содержание списка дочерних устройств.

Образцы драйверов KbFiltr, Osrusbf2/EnumSwitches и Toaster Bus создают объекты PDO и поддерживают статическое и динамическое перечисление устройств.

## **Драйверы унаследованных устройств и объекты устройств управления (KMDF)**

Кроме функциональных драйверов, драйверов шины и драйверов фильтра для устройств Plug and Play, инфраструктура поддерживает разработку драйверов для унаследованных устройств, которые не управляются моделью времени жизни Plug and Play. Такие драйверы создают объекты устройств управления, не являющиеся частью стека устройств Plug and Play. Менеджер ввода/вывода доставляет запросы объекту устройства управления, не посыпая их дальше вниз по стеку устройств.

Драйверы Plug and Play также могут использовать объекты устройств управления для реализации интерфейсов управления, работающих независимо от стека устройств. Приложение может посыпать запросы непосредственно объекту устройства управления, таким образом, минуя любые операции фильтрации, осуществляемые другими драйверами в стеке.

Например, драйвер фильтра, установленный под другим драйвером фильтра в стеке устройств, может создать объект устройства управления для того, чтобы гарантировать получение всех запросов IOCTL, направленных базовому устройству. Драйвер, работающий в стеке устройств, который содержит драйвер порта или класса, не разрешающий специальных запросов IOCTL, может создать объект устройства управления, чтобы получать такие запросы.

Объект устройства управления обычно организует очередь, и драйвер может пересыпать запросы с этой очереди объекту устройства Plug and Play с помощью исполнителя ввода/вывода.

Так как объекты устройств управления не являются частью стека устройств Plug and Play, драйвер должен извещать инфраструктуру о завершении их инициализации, вызывая для этого метод `WdfControlFinishInitializing`. Кроме этого, драйвер должен самостоятельно удалить объект устройства при удалении устройства, т. к. только он знает, каким образом управлять временем жизни данного объекта.

Объекты устройств управления создаются образцами драйверов NdisProt, NonPnP и Toaster Filter.

### **Совет**

В главе 24 рассматривается вопрос, каким образом вставлять примечания в функции обратного вызова драйвера (чтобы SDV мог анализировать их на удовлетворение правилам KMDF), требующие выполнения метода `WdfControlFinishInitializing` для объекта устройства управления, а также ликвидации объекта устройства управления, созданного драйвером, в должное время.

## Драйверы WDF, типы драйверов и типы объектов устройств

Типичное устройство Plug and Play имеет один функциональный драйвер и один драйвер шины, но может иметь любое количество драйверов фильтра. Функциональный драйвер является основным драйвером устройства. Драйвер шины перечисляет свое устройство и любые другие устройства, подключенные к даннойшине или контроллеру. Драйверы фильтров модифицируют один или несколько типов запросов ввода/вывода для стека устройств.

Драйвер UMDF обычно создает один объект устройства для каждого управляемого им устройства. Он может работать в качестве драйвера фильтра или функционального драйвера устройства. В настоящее время UMDF не поддерживает драйверов шины.

Функциональный драйвер или драйвер фильтра UMDF обычно создает один объект устройства для каждого управляемого им устройства. Драйвер шины KMDF обычно является функциональным драйвером для своего устройства и драйвером шины для устройств, перечисляемых его устройством. Поэтому он создает объект FDO для своего устройства и по объекту PDO для каждого перечисленного устройства. Например, драйвер хаба USB работает в качестве функционального драйвера для самого хаба и в качестве драйвера шины для каждого устройства USB, подключенного к его шине. Поэтому он создает объект FDO для хаба и по объекту PDO для каждого подключенного к хабу устройства. После перечисления объектов PDO драйвером шины, менеджер PnP загружает функциональный драйвер для каждого подключенного устройства USB, после чего функциональный драйвер создает объекты FDO для устройств.

Любой драйвер KMDF может также создавать один или несколько объектов устройств управления, хотя эта возможность применяется в очень немногих драйверах.

### Примечание

Windows направляет запросы конкретным объектам устройств, а не драйверам. Некоторые типы запросов применимы только к объектам FDO или FiDO, а другие — только к объектам PDO. Поэтому некоторые типы обратных вызовов применимы только к объектам FDO или FiDO, а другие — только к объектам PDO.

Драйверы, которые могут создавать несколько типов объектов устройств — например, драйвер шины, создающий объект FDO и один или несколько объектов PDO — обычно должны реализовать некоторые обратные вызовы только для объектов FDO и некоторые вызовы только для объектов PDO, особенно для возможностей Plug and Play и управление энергопотреблением.

В этой книге такие обратные вызовы называются вызовами, специфичными для объектов FDO, и вызовами, специфичными для объектов PDO, соответственно, подобно тому, как это делает сама WDF, а не обратными вызовами для функциональных драйверов или для драйверов шины.

## Свойства устройств

Каждый объект устройства содержит информацию о свойствах своего устройства. Инфраструктура устанавливает значения по умолчанию для свойств на основе информации, которую она запрашивает как у системы, так и у драйвера шины. Драйвер может изменить некоторые из этих значений, когда он создает и инициализирует объект устройства. Свойства устройств, поддерживаемые WDF, приводятся в табл. 6.2.

Таблица 6.2. Свойства устройств

Свойство	Описание	Поддерживающая инфраструктура
Требуемое выравнивание	Требуемое выравнивание адреса устройства для операций передачи содержимого памяти	KMDF
Список стандартных потомков	Список стандартных потомков, которые инфраструктура создает для объекта FDO (только для объекта FDO)	KMDF, только для объекта FDO
Стандартная очередь ввода/вывода	Очередь ввода/вывода, которая получает запросы ввода/вывода устройства, если только драйвер не создает дополнительные очереди ввода/вывода	UMDF (посредством файла INF) и KMDF
Характеристики устройства	Набор характеристик устройства, которые сохраняются в виде флагов. Драйвер может установить все эти характеристики одновременно или некоторые характеристики индивидуально	UMDF; KMDF
Идентификатор экземпляра устройства	Строка, представляющая идентификатор экземпляра устройства	UMDF; KMDF
Имя устройства	Имя объекта устройства Down (UMDF) или объекта устройства, создаваемого драйвером (KMDF). Клиенты могут использовать имя объекта устройства KMDF, но не имя устройства UMDF, чтобы открыть устройство	UMDF; KMDF
Состояние устройства	Рабочее состояние устройства Plug and Play	UMDF; KMDF
Фильтр	Булево значение, указывающее, представляет ли объект устройства драйвер фильтра	UMDF; KMDF
Стандартный исполнитель ввода/вывода	Следующий нижележащий драйвер в стеке устройств, которому драйвер пересыпает запросы ввода/вывода	UMDF; KMDF
Родитель	Объект FDO родительской шины для объекта перечисленного дочернего устройства (только PDO)	KMDF, только для объекта FDO
Возможности Plug and Play	Возможности Plug and Play устройства, например, обработка блокировки, неожиданного удаления, и подобные возможности	UMDF; KMDF
Состояние Plug and Play	Текущее состояние машины состояний Plug and Play инфраструктуры	KMDF
Возможности энергопотребления	Возможности управления энергопотреблением устройства, включая промежуточные состояния энергопотребления, состояния, из которых устройство может активировать сигнал пробуждения, и родственные возможности	KMDF
Владение политикой энергопотребления	Булево значение, указывающее, является ли объект устройства владельцем политики энергопотребления для устройства	UMDF; KMDF
Состояние политики энергопотребления	Текущее состояние инфраструктурной машины состояний политики энергопотребления	KMDF
Состояние энергопотребления	Текущее состояние инфраструктурной машины состояний управления энергопотреблением	KMDF

Таблица 6.2 (окончание)

Свойство	Описание	Поддерживающая инфраструктура
Модель синхронизации (также называется граница блокировки)	Уровень, на котором инфраструктура активирует некоторые обратные вызовы объектов одновременно	UMDF; KMDF

Ни UMDF, ни KMDF не предоставляют всех свойств, которые менеджер ввода/вывода Windows сохраняет для устройства. Но как драйверы UMDF, так и драйверы KMDF могут делать вызовы за пределы инфраструктуры, чтобы запросить значение многих из этих свойств. Драйверы KMDF вызывают функцию режима ядра `IoGetDeviceProperty`, а драйверы UMDF вызывают функции семейства `SetupDiXXX`.

Каким образом делать вызовы за пределы инфраструктур, описывается в *главе 14*. Здесь же приводится пример драйвера UMDF, в котором выполняется запрос о свойствах устройства с помощью вызовов функций семейства `SetupDiXXX`.

Дополнительную информацию о функциях семейства `SetupDiXXX` см. в разделе **Using Device Installation Functions** (Использование функций установки устройств) MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=82107>.

## Инициализирование объекта устройства

Объект устройства содержит информацию об устройстве и поддерживает обратные вызовы для связанных с устройством событий, например, запросов Plug and Play или управления энергопотреблением. Обе инфраструктуры реализуют функции, которые драйвер вызывает для инициализации таких данных для объекта устройства.

- ◆ Инфраструктура UMDF предоставляет эти методы в интерфейсе `IWDFDeviceInitialize`. Инфраструктура передает указатель на этот интерфейс как параметр метода `IDriverEntry::OnDeviceAdd`.
- ◆ Инфраструктура KMDF предоставляет для этого функции семейства `WdfDeviceInitXXX`. Функция обратного вызова `EvtDriverDeviceAdd` получает указатель на структуру `WDFDEVICE_INIT`, которая заполняется функциями семейства `WdfDeviceInitXXX`.

Драйвер вызывает функции инициализации объекта устройства перед тем, как он создаст объект устройства, в ходе выполнения функции обратного вызова для события добавления устройства.

С помощью этих функций драйвер может инициализировать возможности Plug and Play устройства и модель параллелизма, которая используется, кроме прочего, с функциями обратного вызова для событий ввода/вывода для объекта устройства. Драйверы режима ядра имеют более обширные возможности доступа к аппаратной части устройства, чем драйверы пользовательского режима, а также могут поддерживать дополнительные возможности. Поэтому объекты устройств драйверов KMDF обычно требуют большего объема инициализации, чем объекты устройств драйверов UMDF.

## Объекты очередей и другие вспомогательные объекты

После создания и инициализации драйвером объекта устройства, драйвер создает дополнительные объекты, ассоциированные с объектом устройства. Эти дополнительные объекты

предоставляют сервисы, используемые драйвером для управления устройством и обработки запросов ввода/вывода. Совместно с объектами устройства они составляют рабочую структуру драйвера.

Наиболее распространенные дополнительные объекты, создаваемые драйверами, перечислены в табл. 6.3. Каждый из приведенных в таблице объектов обычно является потомком объекта устройства.

**Таблица 6.3. Наиболее употребляемые объекты**

Объект	Описание	Поддерживающая инфраструктура
Очередь ввода/вывода	Управляет потоком запросов ввода/вывода к драйверу	UMDF и KMDF (см. главу 8)
Получатель ввода/вывода	Представляет исполнителя запросов ввода/вывода	UMDF и KMDF (см. главу 9)
Устройство, интерфейс и канал USB	Представляет устройство USB и описывает конфигурацию USB и конечные точки в конфигурации	UMDF и KMDF (см. главу 9)
Прерывание	Представляет вектор прерывания или сообщение прерывания	KMDF (см. главу 16)
Блокировка	Предоставляет сериализацию для разделяемых ресурсов	KMDF (см. главу 10)
Поставщик и экземпляр интерфейса WMI	Предоставляет возможности интерфейса WMI, чтобы драйвер мог экспортить информацию другим компонентам	KMDF (см. главу 12)
Выключатель (enabler) DMA, транзакция DMA и общий буфер	Разрешает использование возможностей DMA инфраструктуры и описывает транзакцию и буфер DMA	KMDF (см. главу 17)

В этой главе указывается, когда драйверы должны создавать и инициализировать эти объекты, но не предоставляется подробного объяснения об их назначении и работе.

## Интерфейсы устройств

Приложения пользовательского режима и системные компоненты, которые посылают запросы ввода/вывода устройству, должны быть способными идентифицировать устройство, чтобы они могли открыть его. Возможности идентификации предоставляются интерфейсом устройства.

Интерфейс устройства представляет собой логическое группирование поддерживаемых устройством операций ввода/вывода. Каждому классу интерфейса устройства присваивается глобально уникальный идентификатор (GUID). Компания Microsoft предоставляет идентификаторы для большинства классов интерфейса устройства в специфичных для устройства заголовочных файлах; если необходимо, поставщики могут предоставлять определения идентификаторов GUID для своих специфических классов интерфейса.

Дополнительно по этому вопросу см. раздел **Using GUIDs in Drivers** (Использование идентификаторов GUID в драйверах) на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82109>.

Драйвер регистрирует свое устройство членом класса интерфейса устройства, создавая экземпляр интерфейса устройства и предоставляемый соответствующий идентификатор GUID и необязательную строку. Эта строка однозначно идентифицирует созданный экземпляр интерфейса, с тем чтобы, если драйвер создаст еще один экземпляр этого же класса интерфейса, он мог бы определить, какой интерфейс используется определенным клиентом, таким образом зная, куда направлять запросы ввода/вывода клиента.

Драйвер может создать интерфейс в любое время после создания объекта устройства, но обычно он это делает в функции обратного вызова при событии добавления устройства. По умолчанию инфраструктура разрешает интерфейсы устройства, когда устройство переходит в рабочее состояние, и запрещает их, когда устройство выходит из рабочего состояния.

Тестовые приложения, предоставленные с образцами драйверов Osrusbf2 и Fx2\_Driver, демонстрируют, как клиентское приложение может использовать интерфейс устройства для нахождения устройства.

## Интерфейсы устройств и символьные ссылки

Клиент может идентифицировать и открыть устройство, используя либо символьную ссылку, либо интерфейс устройства. Для приложений символьные ссылки более просты в применении, но в драйверах с ними другая история.

В драйверах символьные ссылки представляют угрозу безопасности; кроме этого, их трудно сохранять при перезагрузках и переконфигурировании оборудования. С точки зрения драйвера, использование интерфейсов устройств намного проще. Интерфейсы устройств являются постоянными, поэтому приложение может сохранять их в своих конфигурационных данных и использовать в дальнейшем для нахождения устройства. Кроме этого, при удалении устройства система автоматически выполняет очистку его интерфейса.

## Создание и инициализирование объекта устройства UMDF

Драйверы UMDF создают объект обратного вызова устройства, который работает в паре с инфраструктурным объектом устройства. Объект обратного вызова устройства реализует интерфейсы обратного вызова, с помощью которых инфраструктура уведомляет драйвер о событиях Plug and Play и о файловых событиях.

Метод `IDriverEntry::OnDeviceAdd` объекта обратного вызова драйвера создает и инициализирует объект устройства. Инфраструктура вызывает этот метод со следующими параметрами:

- ◆ указатель на интерфейс `IWDFDriver` на объекте драйвера;
- ◆ указатель на интерфейс `IWDFDeviceInitialize` на объекте драйвера.

Интерфейс `IWDFDriver` определяет метод `CreateDevice`, который вызывается драйвером для создания инфраструктурного объекта устройства. Интерфейс `IWDFDeviceInitialize` содержит несколько методов, с помощью которых драйвер может инициализировать объект устройства.

Метод `OnDeviceAdd`:

- ◆ создает объект обратного вызова устройства;
- ◆ создает и инициализирует инфраструктурный объект устройства;
- ◆ создает очереди ввода/вывода, ассоциированные с объектом устройства.

В функциональном драйвере метод `OnDeviceAdd` также должен создать и разрешить интерфейс устройства для того, чтобы клиенты могли посыпать ввод/вывод устройству.

### Примечание

В образце драйвера `Fx2_Driver` метод `OnDeviceAdd` вызывает несколько открытых методов на объекте обратного вызова устройства для выполнения этих задач. А именно:

- метод `CMyDevice::CreateInstance` создает объект обратного вызова устройства;
- метод `CMyDevice::Initialize` создает и инициализирует инфраструктурный объект устройства;
- метод `CMyDevice::Configure` создает очереди ввода/вывода и интерфейс устройства.

Исходный код этого драйвера находится в файле `Device.cpp`. Вместо того чтобы дублировать все эти методы, в этой главе приводятся фрагменты кода, в которых демонстрируется, как выполнить каждую задачу.

## Создание объекта обратного вызова устройства

Для создания объекта обратного вызова драйвер UMDF просто использует оператор `new`, как показано в следующем примере:

```
PCMyDevice device;
device = new CMyDevice();
if (NULL == device) {
    return E_OUTOFMEMORY;
}
```

Если создание объекта завершается неудачей, драйвер возвращает ошибку нехватки памяти.

## Создание и инициализирование инфраструктурного объекта устройства

Интерфейс `IWDFDeviceInitialize` реализует методы, которые инициализируют свойства устройства и другие установки и возвращают информацию об устройстве. Эти методы перечислены в табл. 6.4.

**Таблица 6.4. Методы интерфейса `IWDFDeviceInitialize`**

Метод	Описание
<code>AutoForwardCreateCleanupClose</code>	Указывает, должна ли инфраструктура пересыпать запросы на создание, очистку и закрытие стандартному исполнителю ввода/вывода, если драйвер не имеет функции обратного вызова для таких запросов. По умолчанию инфраструктура пересыпает запросы драйверам фильтров, но не пересыпает их функциональным драйверам (см. главу 8)

Таблица 6.4 (окончание)

Метод	Описание
GetPnpCapability	Возвращает WdfTrue или WdfFalse, чтобы указать, поддерживает или нет устройство одну из следующих возможностей Plug and Play: <ul style="list-style-type: none"> <li>• WdfPnpCapLockSupported — устройство может быть заблокировано в своем разъеме, чтобы предотвратить его выброс. Не применимо к носителям;</li> <li>• WdfPnpCapLockSupported — можно осуществить выброс устройства из его разъема. Не применимо к носителям;</li> <li>• WdfPnpCapRemovable — устройство можно извлечь при работающей системе;</li> <li>• WdfPnpCapDockDevice — устройство является док-станцией;</li> <li>• WdfPnpCapSurpriseRemovalOk — устройство может быть извлечено пользователем без применения системной утилиты для безопасного удаления устройства;</li> <li>• WdfPnpCapNoDisplayInUI — устройство можно запретить для отображения в Диспетчере устройств</li> </ul>
RetrieveDeviceInstanceId	Возвращает строку, представляющую идентификатор экземпляра устройства. Идентификатор экземпляра обозначает данный конкретный экземпляр устройства. Драйвер может использовать идентификатор экземпляра в вызовах функций семейства SetupDiXXX для получения дополнительных свойств устройства
RetrieveDevicePropertyStore	Возвращает хранилище свойств устройства, посредством которого драйвер может считывать и записывать данные в реестр (см. главу 12)
SetFilter	Указывает, представляет ли объект устройства драйвер фильтра. Инфраструктура передает вниз по стеку все запросы ввода/вывода, которые не отфильтровываются драйвером
SetLockingConstraint	Устанавливает модель синхронизации для методов обратного вызова объекта устройства. Значение по умолчанию — WdfDeviceLevel (см. главу 10)
SetPnpCapability	Разрешает, запрещает или выбирает установки по умолчанию для определенной возможности Plug and Play устройства, указанного в методе GetPnpCapability
SetPowerPolicyOwnership	Извещает инфраструктуру, что объект устройства является владельцем политики энергопотребления для устройства. Значение по умолчанию — FALSE (см. главу 7)

Драйвер вызывает один или несколько из этих методов для установки характеристик устройства перед тем, как создать объект устройства. Драйвер фильтра должен вызывать метод SetFilter, а владелец политики энергопотребления должен вызывать метод SetPowerPolicyOwnership. В противном случае никакие из этих установок не требуются.

После установки драйвером характеристик устройства с помощью методов интерфейса IWDFDeviceInitialize драйвер создает инфраструктурный объект устройства с помощью метода IWDFDriver::CreateDevice. Этот метод принимает следующие параметры:

- ◆ указатель на интерфейс IWDFDeviceInitialize, который был передан драйверу;
- ◆ указатель на интерфейс IUnknown объекта обратного вызова устройства для драйвера;

- ◆ адрес, по которому возвратить указатель на интерфейс IWDFDevice инфраструктурного объекта устройства.

Если инфраструктура успешно создает инфраструктурный объект устройства, она освобождает ссылку, добавленную методом CreateDevice на возвращенный интерфейс. Эта ссылка не является необходимой для обеспечения постоянства объекта.

В листинге 6.3 приводится фрагмент кода, демонстрирующий инициализирование характеристик и создание инфраструктурного объекта устройства образом драйвера Fx2\_Driver.

#### Листинг 6.3. Инициализирование характеристик и создание объекта устройства инфраструктуры UMDF

```
FxDeviceInit->SetLockingConstraint(WdfDeviceLevel);
{
    IUnknown *unknown = device->QueryIUnknown();
    hr = FxDriver->CreateDevice(FxDeviceInit, unknown, &fxDevice);
    unknown->Release();
}
if (S_OK == hr) {
    fxDevice->Release();
}
```

Образец драйвера Fx2\_Driver устанавливает модель блокировки (которая также называется областью синхронизации) для драйвера, вызывая метод SetLockingConstraint интерфейса IWDFDeviceInitialize. Модель блокировки определяет, устанавливает ли инфраструктура блокировку перед обратным вызовом функций для определенных событий ввода/вывода и файловых операций. Значение WdfDeviceLevel означает, что инфраструктура блокирует объект устройства, чтобы предотвратить одновременное выполнение этих функций для любых очередей или файлов, ассоциированных с объектом устройства. Модель блокировки не охватывает обратные вызовы для событий Plug and Play, энергопотребления и завершения ввода/вывода. Синхронизация является, наверное, самым трудным моментом в реализации драйвера, поэтому необходимо быть уверенным в том, что вы полностью понимаете, что влечет за собой применение выбранной модели блокировки.

Подробно вопрос области синхронизации рассматривается в главе 10.

После установки драйвером характеристик устройства, он создает инфраструктурный объект устройства, вызывая для этого метод IWDFDriver::CreateDevice.

## Пример UMDF: интерфейс устройства

Драйвер UMDF создает экземпляр класса интерфейса устройства для своего устройства с помощью метода IWDFDevice::CreateDeviceInterface. При успешном создании экземпляра инфраструктура автоматически разрешает интерфейс при переходе устройства в рабочее состояние и запрещает его при выходе устройства из рабочего состояния. В листинге 6.4 показан исходный код для регистрации драйвером Fx2\_Driver класса интерфейса своего устройства.

#### Листинг 6.4. Создание интерфейса устройства UMDF

```
hr = m_FxDevice->CreateDeviceInterface (
    &CUID_DEVINTERFACE_OSRUSBFX2, NULL);
```

Как можно видеть в листинге 6.4, драйвер Fx2\_Driver создает интерфейс, вызывая метод CreateDeviceInterface. Он передает в параметрах свой идентификатор GUID класса интерфейса, определенный в заголовочном файле WUDFOsrUsbPublic.h, и справочную строку NULL.

Инфраструктура автоматически разрешает интерфейс драйвера.

Драйвер может запретить интерфейс устройства, чтобы указать, что устройство больше не принимает запросов. Драйвер запрещает интерфейс устройства, вызывая метод IWDFDevice::AssignDeviceInterfaceState и передавая ему булево значение FALSE в качестве параметра. На практике драйверы запрещают интерфейс нечасто. При удалении устройства инфраструктура автоматически запрещает интерфейс драйвера.

## Создание и инициализирование объекта устройства KMDF

Любой драйвер KMDF, поддерживающий Plug and Play, должен иметь функцию обратного вызова *EvtDriverDeviceAdd*. Функция *EvtDriverDeviceAdd* отвечает за создание и инициализирование объекта устройства и связанных с ним ресурсов. Инфраструктура вызывает функцию обратного вызова *EvtDriverDeviceAdd*, когда система обнаруживает устройство, контролируемое драйвером.

Функция обратного вызова *EvtDriverDeviceAdd* создает объект WDFDEVICE, который представляет устройство и выполняет множественные задачи инициализации, чтобы предоставить инфраструктуре информацию, необходимую ей для организации своих внутренних структур.

Для этого функция *EvtDriverDeviceAdd* выполняет такую последовательность операций:

1. Заполняет структуру инициализации устройства информацией, необходимой для создания объекта устройства.
2. Организует область контекста объекта устройства.
3. Создает объект устройства.
4. Выполняет дополнительные работы по инициализации и запуску устройства, например, создает очереди ввода/вывода и интерфейс устройства.

Драйвер шины, перечисляющий дочерние устройства, обычно создает несколько объектов устройств: объект FDO для самой шины и по одному объекту PDO для каждого дочернего устройства, подключенного к шине. После создания драйвером шины объектов PDO система загружает функциональные драйверы для дочерних устройств, и инфраструктура вызывает их функции обратного вызова *EvtDriverDeviceAdd*, которые, в свою очередь, создают объекты FDO для дочерних устройств.

## Структура инициализации устройства KMDF

В отличие от большинства других объектов KMDF, объект устройства не имеет структуры конфигурации. Вместо этого, для конфигурирования объекта устройства драйвер использует структуру *WDFDEVICE\_INIT*. Инфраструктура вызывает функцию обратного вызова *EvtDriverDeviceAdd* с указателем на структуру *WDFDEVICE\_INIT* в качестве параметра, а драйвер заполняет эту структуру информацией об устройстве и драйвере, вызывая методы семейства

`WdfDeviceInitXXX`. Инфраструктура использует эту информацию позже, когда она создает объект `WDFDEVICE`.

Драйвер может вызывать методы семейства `WdfDeviceInitXXX` для выполнения следующих задач:

- ◆ установки характеристик устройства;
- ◆ установки типа ввода/вывода;
- ◆ создания области контекста или установки других атрибутов для запросов ввода/вывода, направляемых инфраструктурой драйверу;
- ◆ регистрации обратных вызовов функций для событий Plug and Play и управления энергопотреблением;
- ◆ регистрации обратных вызовов для событий создания, закрытия и очистки файла.

Кроме этого, для объектов FDO, FiDO и PDO выполняются специфические операции инициализации. Инициализация объектов FDO и FiDO описывается в следующих разделах. Для объектов PDO существуют специальные требования; см. разд. "Перечисление дочерних устройств (только PDO-объекты KMDF)" далее в этой главе.

## Инициализация KMDF объектов FDO

Объект FDO является владельцем по умолчанию политики энергопотребления для своего устройства. Если устройство может бездействовать в состоянии низкого энергопотребления или генерировать сигнал пробуждения, драйвер обычно регистрирует для объекта FDO функции обратного вызова для событий политики энергопотребления, вызывая для этого метод `WdfDeviceInitSetPowerPolicyEventCallbacks`.

Объекты FDO могут добавлять и удалять ресурсы со списка требуемых ресурсов по информации, предоставленной драйвером. Драйвер KMDF реализует функции обратного вызова `EvtDeviceFilterRemoveResourceRequirements` и `EvtDeviceFilterAddResourceRequirements` и регистрирует их с помощью метода `WdfFdoInitSetEventCallbacks`.

Если драйвер добавляет ресурсы в список требуемых ресурсов, он должен удалить эти добавленные ресурсы с окончательного списка назначенных ресурсов перед запуском устройства. Для этой задачи драйвер KMDF предоставляет функцию обратного вызова `EvtDeviceRemoveAddedResources`. Драйверы для унаследованных устройств иногда добавляют или удаляют ресурсы, но драйверы для современных устройств Plug and Play делают это нечасто.

Другие методы семейства `WdfFdoInitXXX` извлекают свойства устройства, возвращают указатель на PDO-объект WDM для стека устройств, предоставляют доступ к ключу реестра устройства и выполняют другие операции, специфичные для объекта FDO.

## Инициализация KMDF объектов FiDO

С помощью метода `WdfFdoInitSetFilter` драйвер информирует инфраструктуру о том, что объект устройства представляет фильтр. Этот метод заставляет инфраструктуру изменить свои стандартные установки для обработки запросов ввода/вывода, не обрабатываемых драйвером. Когда прибывает запрос ввода/вывода, не обрабатываемый объектом FiDO, вместо того, чтобы не выполнять этот запрос, что является стандартным действием для объектов FDO и PDO, инфраструктура передает этот запрос следующему нижележащему драйверу. Для объектов FiDO драйвер может вызывать те же самые методы семейства `WdfDeviceInitXXX`, что и для объектов FDO.

## Область контекста объекта устройства

Драйверы обычно хранят данные, имеющие отношение к объекту устройства, с самим объектом. Для этой цели драйверы KMDF применяют область контекста объекта. Область контекста объекта выделяется из нестраничного пула; ее организация определяется драйвером. Когда драйвер KMDF создает объект устройства, он обычно инициализирует область контекста и указывает ее размер и тип. Когда инфраструктура удаляет объект, она также удаляет его область контекста.

В листинге 6.5 показано определение драйвером Osrusbf2 области контекста для объекта своего устройства. Этот исходный код находится в заголовочном файле Osrusbf2.h.

### Листинг 6.5. Определение области контекста объекта устройства

```
typedef struct _DEVICE_CONTEXT {
    WDFUSBDEVICE                         UsbDevice;
    WDFUSBINTERFACE                       UsbInterface;
    WDFUSBPIPE                            BulkReadPipe;
    WDFUSBPIPE                            BulkWritePipe;
    WDFUSBPIPE                            InterruptPipe;
    UCHAR                                 CurrentSwitchState;
    WDFQUEUE                             InterruptMsgQueue;
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, GetDeviceContext)
```

Как можно видеть в предыдущем листинге, в заголовочном файле сначала определяется область контекста типа `DEVICE_CONTEXT`, после чего вызывается макрос `WDF_DECLARE_CONTEXT_TYPE_WITH_NAME`. Этот макрос создает метод доступа, который ассоциируется с типом контекста. Таким образом, когда выполняется обратный вызов функции `EvtDriverDeviceAdd` драйвера Osrusbf2, метод доступа `GetDeviceContext` для чтения и записи в область контекста типа `DEVICE_CONTEXT` уже создан.

Для ассоциирования именованной области контекста с объектом драйвер должен инициализировать структуру атрибутов объекта информацией об области контекста, вызывая для этого макрос `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE` из функции обратного вызова `EvtDriverDeviceAdd`.

## Создание объектов устройств KMDF

После вызова драйвером KMDF методов семейства `WdfDeviceInitXxx` для заполнения структуры `WDFDEVICE_INIT` он устанавливает атрибуты для объекта устройства в структуре атрибутов. Атрибуты для объекта устройства почти всегда включают размер и тип области контекста и часто включают функцию обратного вызова очистки объекта; также может быть включена область синхронизации.

Драйвер передает структуру атрибутов и структуру `WDFDEVICE_INIT` в качестве параметров методу `WdfDeviceCreate` для создания объекта устройства. Метод `WdfDeviceCreate` создает объект `WDFDEVICE`, подсоединяет объект устройства к стеку устройств Plug and Play и возвращает дескриптор объекта.

Структура атрибутов обсуждается в главе 5.

## Дополнительные задачи, выполняемые функцией *EvtDriverDeviceAdd*

После того как функция обратного вызова *EvtDriverDeviceAdd* создаст объект устройства, она должна выполнить следующие задачи.

- ◆ Если объект устройства является владельцем политики энергопотребления, установить настройки политики бездействия и пробуждения.

Подробности см. в главе 7.

- ◆ Зарегистрировать функции обратного вызова для событий ввода/вывода и создать очереди ввода/вывода для объекта устройства.

Дополнительную информацию по этому вопросу см. в главе 8.

- ◆ Если требуется, создать интерфейс устройства с помощью метода *WdfDeviceCreateDeviceInterface*.

- ◆ Если устройство поддерживает прерывания, создать объект прерываний.

Прерывания рассматриваются в главе 16.

- ◆ Создать объекты WMI.

Руководящие принципы по данному вопросу приводятся в главе 12.

Инфраструктура запускает очереди и подключает объект прерываний позже, в процессе запуска устройства.

## Пример KMDF: функция обратного вызова *EvtDriverDeviceAdd*

В листинге 6.6 приведен исходный код сокращенной версии функции обратного вызова *EvtDriverDeviceAdd* из драйвера Osrusbf2. Объяснение пронумерованных компонентов дается после листинга.

### Листинг 6.6. Функция обратного вызова *EvtDriverDeviceAdd*

```
NTSTATUS OsrFxEvtDeviceAdd(
    IN WDFDRIVER           Driver,
    IN PWDDEVICE_INIT       DeviceInit
)
{
    WDF_PNPPOWER_EVENT_CALLBACKS      pnpPowerCallbacks;
    WDF_OBJECT_ATTRIBUTES             attributes;
    NTSTATUS                         status;
    WDFDEVICE                        device;
    WDF_DEVICE_PNP_CAPABILITIES     pnpCaps;
    UNREFERENCED_PARAMETER(Driver);
    PAGED_CODE();
    // Инициализация структуры pnpPowerCallbacks.
    . . . // Код опущен.
    // [1] Регистрация обратных вызовов функций для событий
    //     Plug and Play и управления энергопотреблением.
    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);
```

```
//[2] Указываем тип ввода/вывода, выполняемый данным драйвером.
WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoBuffered);
//[3] Инициализируем структуру атрибутов объекта.
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, DEVICE_CONTEXT);
//[4] Вызываем инфраструктуру для создания объекта устройства.
status = WdfDeviceCreateC&Ddevicelnit, &attributes, &device);
if (!NT_SUCCESS(status)) {
    return status;
}
// Устанавливаем возможности PnP-устройства, конфигурируем
// и создаем очереди ввода/вывода.
. . . // Код опущен.
//[5] Регистрация интерфейса устройства.
status = WdfDeviceCreateDeviceInterface(device,
                                         &CUID_DEVINTERFACE_OSRUSBFX2,
                                         NULL); // Справочная строка.
if (!NT_SUCCESS(status)) {
    return status;
}
return status;
}
```

Функция `EvtDriverDeviceAdd` в листинге 6.6 выполняет такую последовательность операций:

1. Регистрирует обратные вызовы для событий Plug and Play и управления энергопотреблением.

Драйвер инициализирует структуру (не показанную в листинге) информацией о функциях обратного вызова для обработки событий Plug and Play и управления энергопотреблением и вызывает метод `WdfDeviceInitSetPnpPowerEventCallbacks`, чтобы сохранить эту информацию в структуре `WDFDEVICE_INIT`.

2. Устанавливает тип ввода/вывода.

Драйвер указывает тип ввода/вывода для запросов на чтение и запись — буферизованный, прямой или и непрямой, и небуферизованный. Устанавливается стандартный тип `WdfDeviceIoBuffered`.

3. Инициализирует структуру атрибутов объекта.

Драйвер инициализирует структуру атрибутов объекта типом области контекста, чтобы инфраструктура могла создать область контекста для объекта устройства.

4. Создает объект устройства.

Драйвер вызывает метод `WdfDeviceCreate` для создания объекта устройства. Он передает в качестве параметров структуру `WDFDEVICE_INIT` и структуры атрибутов объекта, который он заполнил, а метод возвращает ему дескриптор объекта устройства.

5. Создает интерфейс устройства.

Для создания интерфейса устройства драйвер вызывает метод `WdfDeviceCreateInterface`. Драйвер передает методу в качестве параметров дескриптор объекта устройства, указатель на идентификатор GUID и указатель на необязательную справочную строку. Идентификатор GUID определен в заголовочном файле `Public.h`. Драйвер может использовать справочную строку для различия между двумя или несколькими устройствами одного и того же класса интерфейса, т. е. между двумя или несколькими устройствами с одинако-

выми идентификаторами GUID. В параметре справочной строки драйвер Osrusbfx2 передает NULL.

По умолчанию инфраструктура разрешает интерфейсы устройства, когда устройство переходит в рабочее состояние, и запрещает их, когда устройство выходит из рабочего состояния. Поэтому в большинстве случаев драйверу не нужно разрешать или запрещать интерфейс.

## Перечисление дочерних устройств (только PDO-объекты KMDF)

Инфраструктура KMDF поддерживает как статическую, так и динамическую модель перечисления дочерних устройств. Она также включает некоторые специфичные для объектов PDO возможности. Эти возможности и рассматриваются в данном разделе.

### Статическое и динамическое перечисление в драйверах шины

После создания другим драйвером объекта PDO для шины, инфраструктура активизирует функцию обратного вызова *EvtDriverDeviceAdd* драйвера шины. Драйвер шины создает объект FDO для самого устройства, а потом перечисляет дочерние устройства, подключенные к его устройству, и создает по объекту PDO для каждого из них. Драйвер может перечислять дочерние устройства или статически, или динамически.

Драйвер KMDF предоставляет информацию о своих потомках в объекте *WDFCHILDLIST*, а инфраструктура предоставляет информацию из этого объекта менеджеру PnP в ответ на его запрос.

Если драйвер вызывает метод *WdfFdoInitSetDefaultChildListConfig*, инфраструктура создает пустой стандартный объект *WDFCHILDLIST* и устанавливает объект FDO в качестве его родителя. Объект списка дочерних устройств содержит информацию о дочерних устройствах, перечисленных родительским устройством.

### Динамическое перечисление

При динамическом перечислении драйвер должен, как минимум, выполнить следующие задачи.

- ◆ Вызвать метод *WdfFdoInitSetDefaultChildListConfig* из функции обратного вызова *EvtDriverDeviceAdd*, чтобы сконфигурировать список дочерних устройств и зарегистрировать функции обратного вызова семейства *EvtChildListXxx*, перед тем, как он создаст объект FDO для шины.
- ◆ После создания объекта FDO перечислить дочерние устройства и заполнить список дочерних устройств, применяя для этого методы семейства *WdfChildListXxx*.  
Драйвер может выполнить эту задачу в любое время после создания объекта устройства.
- ◆ Реализовать функцию обратного вызова *EvtChildListCreateDevice*, которая создает объект PDO для дочернего устройства. При вызове этой функции инфраструктура передает ей в качестве параметра указатель на структуру *WDFDEVICE\_INIT*.  
Функция обратного вызова заполняет эту структуру и вызывает метод *WdfDeviceCreate* для создания объекта PDO для дочернего устройства.

- ◆ Реализовать другие функции обратного вызова семейства `EvtChildListXxx`, требуемые для предоставления поддержки дочерним устройствам устройства.

Драйвер, выполняющий динамическое перечисление, применяет методы семейства `WdfChildListXxx` для внесения изменений в список своих дочерних устройств.

### Статическое перечисление

Драйверу, выполняющему статическое перечисление, не требуется реализовывать никаких функций обратного вызова семейства `EvtChildListXxx`. Вместо этого, его функция обратного вызова `EvtDriverDeviceAdd` должна выполнить следующие задачи:

- ◆ после создания драйвером объекта FDO перечислить дочерние устройства;
- ◆ с помощью метода `WdfPdoInitAllocate` получить указатель на структуру `WDFDEVICE_INIT` для каждого дочернего объекта PDO;
- ◆ инициализировать должным образом структуру `WDFDEVICE_INIT` каждого дочернего устройства и с помощью метода `WdfDeviceCreate` создать дочерний объект PDO;
- ◆ с помощью метода `WdfFdoAddStaticChild` обновить стандартный список дочерних устройств.

В общем, драйверы, выполняющие статическое перечисление, не вносят изменений в список своих дочерних устройств.

Но если необходимо, драйвер может обновить статический список дочерних устройств с помощью методов семейства `WdfFdoXxx`.

## Инициализация объектов PDO

Некоторые функции обратного вызова применяются только для объектов PDO.

Объекты PDO могут поддерживать функции обратных вызовов, которые отвечают на запросы об имеющихся и требуемых ресурсах устройства, на запросы на выполнение механической блокировки или эжекции устройства из его док-станции, а также на запросы на разрешение и запрещение сигнала пробуждения устройства.

Драйверы KMDF регистрируют функции обратного вызова с помощью метода `WdfPdoInitSetEventCallbacks` во время инициализации объекта устройства. В табл. 6.5 приводится список функций обратного вызова, специфичных для объектов PDO.

**Таблица 6.5. Функции обратного вызова для объектов PDO**

Функция	Описание
<code>EvtDeviceDisableWakeAtBus</code>	Конфигурирует шину для запрета одному из ее устройств инициировать сигнал пробуждения
<code>EvtDeviceEject</code>	Обрабатывает операции, связанные с выбросом устройства с его док-станции
<code>EvtDeviceEnableWakeAtBus</code>	Конфигурирует шину для разрешения одному из ее устройств инициировать сигнал пробуждения
<code>EvtDeviceResourceRequirementsQuery</code>	Создает список аппаратных ресурсов, требуемых устройством
<code>EvtDeviceResourcesQuery</code>	Извещает о ресурсах, выделенных устройству при загрузке системы

Таблица 6.5 (окончание)

Функция	Описание
<code>EvtDeviceSetLock</code>	Блокирует устройство, чтобы предотвратить его выброс из док-станции

С помощью других методов семейства `WdfPdoInitXXX` драйвер может указывать данные, специфичные для устройства, например, идентификатор устройства.

## Способы именования устройства для драйверов KMDF

Объекты устройств KMDF не имеют установленных имен. Инфраструктура устанавливает значение константы `FILE_AUTOGENERATED_DEVICE_NAME` в характеристиках устройства для объектов PDO согласно требованиям модели WDM.

Инфраструктура также поддерживает создание и регистрацию интерфейсов устройства на всех устройствах Plug and Play и управляет интерфейсами устройств для своих драйверов.

Во всех возможных случаях следует использовать интерфейсы устройств вместо более старых методов, применяющих установленные имена и символические ссылки.

Но если унаследованные приложения требуют именования устройства, вы можете присвоить ему имя и указать его дескриптор безопасности с помощью языка SDDL (Security Descriptor Definition Language, язык определения дескрипторов безопасности). Дескриптор безопасности определяет пользователей, которые могут открывать устройство.

По соглашению, установленное имя устройства сопоставляется с установленным именем символьной ссылки, например `\DosDevices\MyDeviceName`.

Инфраструктура поддерживает создание символьной ссылки и управление ею и автоматически удаляет ссылку при уничтожении устройства.

Инфраструктура также позволяет создавать именованные символьные ссылки для безымянных устройств Plug and Play.

Кроме рекомендаций WDK, при именовании объектов устройств необходимо придерживаться следующих правил:

- ◆ присваивать имена объектам устройства, только когда это необходимо;
- ◆ предоставлять дескрипторы безопасности для интерфейсов устройств и для именованных объектов устройств.

## Именованные объекты устройств

Хотя приложения открывают устройства по имени, объекты FDO и FiDO обычно не имеют имен. Вместо именования объектов устройств, драйверы определяют интерфейсы устройств, посредством которых приложения могут открывать устройства. Но объектам PDO необходимо присваивать имена.

Информацию об исключениях в именовании см. в разделе **Controlling Device Access in Framework-Based Drivers** (Управление доступом к устройству в драйверах на основе инфраструктуры) WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81579>.

По умолчанию, когда инфраструктура создает объект PDO, менеджер ввода/вывода Windows назначает ему имя, сгенерированное системой. Это стандартное поведение нужно заменять только в тех случаях, когда ваш драйвер поддерживает устаревшее программное обеспечение. Например, когда приложение ожидает определенное имя устройства или если стек драйверов содержит старые драйверы, которым требуются такие имена. Драйвер переопределяет стандартное поведение назначения имен и присваивает имя объекту устройства самостоятельно, с помощью метода `WdfDeviceInitAssignName`.

Каждый именованный объект устройства также должен иметь дескриптор безопасности. С помощью дескриптора безопасности Windows определяет пользователей, которые имеют разрешение на доступ к устройству и его интерфейсам.

Имена символьных ссылок отличаются от имен объектов устройств. Имена символьных ссылок обычно создаются старыми драйверами, которые исполняются с приложениями, использующими для доступа к устройствам имена устройств в стиле MS-DOS. При создании драйвером имени символьной ссылки для своего устройства инфраструктура ассоциирует ссылку с именем, назначенным объекту PDO менеджером ввода/вывода Windows. Но если драйвер присваивает имя объекту FDO, то инфраструктура ассоциирует символьную ссылку с именем объекта FDO.

## Дескрипторы безопасности

Дескриптор безопасности определяет, кто может иметь доступ к определенному ресурсу, например, интерфейсу устройства или объекту устройства. Дескрипторы безопасности создаются с помощью языка SDDL. Несколько полезных для драйверов дескрипторов безопасности определены в заголовочном файле `Wdmsec.h` в WDK.

Как инфраструктура, так и Windows применяют согласно табл. 6.6.

**Таблица 6.6. Стандартные дескрипторы безопасности**

Действие драйвера	Дескриптор безопасности
Создание объекта устройства	Значение по умолчанию, применяемое менеджером ввода/вывода Windows: <code>SDDL_DEVOBJ_SYS_ALL ADM_RWX_WORLD_R_RES_R</code>
Вызов метода <code>WdfDeviceInitAssignName</code>	Значение по умолчанию, применяемое инфраструктурой: <code>SDDL_DEVOBJ_SYS_ALL ADM_ALL</code>
Вызов метода <code>WdfDeviceInitAssignSDDLString</code>	Строка SDDL, передаваемая в параметре при вызове метода

По умолчанию дескриптор безопасности для интерфейса устройства такой же, как и дескриптор безопасности для объекта PDO устройства. Но эта настройка по умолчанию может быть заменена для устройства или класса установки устройства настройкой из файла INF.

Дополнительную информацию о дескрипторах безопасности см. в разделе **SDDL for Device Objects** (Язык SDDL для объектов устройств) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80626>. Также, для получения дополнительной информации о создании настроек безопасности в INF-файлах драйверов см. раздел **Creating Secure Device Installations** (Создание безопасных установочных пакетов устройств) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80625>.

В листинге 6.7 показан фрагмент исходного кода из драйвера Osrusbf2, который назначает строку SDDL "сырому" устройству PDO. Этот код находится в файле Osrusbf2\Sys\Enumswitches\Rawpdo.c.

#### Листинг 6.7. Назначение строки SDDL "сырому" устройству PDO

```
NTSTATUS OsrEvtDeviceListCreatePdo(
    WDFCHILDLIST DeviceList,
    PWDF_CHILD_IDENTIFICATION_DESCRIPTION_HEADER
        IdentificationDescription,
    PWDFDEVICE_INIT ChildInit
)
{
    NTSTATUS status;
    WDFDEVICE hChild = NULL;
    . . . // Дополнительные объявления и код упущены.
    // Для создания "сырого" объекта PDO необходимо
    // предоставить идентификатор GUID класса.
    status = WdfPdoInitAssignRawDevice(ChildInit,
        &CUID_DEVCLASS_OSRUSBFX2);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
    status = WdfDeviceInitAssignSDDLString(ChildInit,
        &SDDL_DEVOBJ_SYS_ALL ADM_ALL);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
    // Выполняем дополнительную инициализацию.
    . . . // Код опущен, чтобы сохранить место.
    // Создаем объект PDO для дочернего устройства.
    status = WdfDeviceCreate(&ChildInit, WDF_NO_OBJECT_ATTRIBUTES,
        &hChild);
    if (!NT_SUCCESS(status)) {
        goto Cleanup;
    }
    // Далее следуют операции дополнительной инициализации и установки.
    . . . // Код опущен, чтобы сохранить место.
Cleanup:
    return status;
}
```

Функция в листинге 6.7 инициализирует специфичную для устройства информацию и создает объект PDO для дочернего устройства USB. Для присвоения дескриптора безопасности она вызывает метод `WdfDeviceInitAssignSDDLString`, передавая в параметрах указатель на структуру `WDFDEVICE_INIT` и указатель на дескриптор безопасности `SDDL_DEVOBJ_SYS_ALL ADM_ALL`. С помощью дескриптора безопасности для объекта PDO Windows определяет, кто имеет права доступа к интерфейсам устройства. Этот дескриптор позволяет управление устройством любому компоненту режима ядра и любому приложению, исполняющемуся с привилегиями администратора. Всем другим пользователям доступ к данному устройству запрещен.

# **ЧАСТЬ III**

## **Применение основ WDF**

- Глава 7.** Plug and Play и управление энергопотреблением
- Глава 8.** Поток и диспетчеризация ввода/вывода
- Глава 9.** Получатели ввода/вывода
- Глава 10.** Синхронизация
- Глава 11.** Трассировка и диагностируемость драйверов
- Глава 12.** Вспомогательные объекты WDF
- Глава 13.** Шаблон UMDF-драйвера

## ГЛАВА 7

# Plug and Play и управление энергопотреблением

Механизм Plug and Play и управление энергопотреблением охватывают множество мероприятий, связанных с установкой, конфигурированием и штатной работой устройств. Приведем лишь несколько примеров ситуаций, требующих поддержки Plug and Play или управления энергопотреблением:

- ◆ подключение пользователем MP3-плеера к работающей системе;
- ◆ неожиданное удаление пользователем флэш-памяти USB;
- ◆ подключение пользователем сетевого кабеля Ethernet при работающей системе;
- ◆ пробуждение спящей системы движением мыши;
- ◆ переход системы в режим сна после определенного периода бездействия.

Для предоставления должной поддержки Plug and Play и управлению энергопотреблением операционная система, драйверы, программы системного администрирования, программы установки устройств, аппаратное обеспечение системы и аппаратное обеспечение устройств должны все работать согласованно.

WDF реализует полностью интегрированную модель для Plug and Play и управления энергопотреблением. Эта модель предоставляет стандартные интеллектуальные настройки, благодаря которым для некоторых драйверов не требуется никакого кода для поддержки простых возможностей Plug and Play и управления энергопотреблением. Поддержку для более сложных возможностей драйверы реализуют с помощью функций обратного вызова. В этой главе даются руководящие принципы для реализации поддержки Plug and Play и управления энергопотреблением в драйверах UMDF и KMDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы</b>	
Fx2_Driver	%wdk%\Src\Umdf\Usb\Fx2_Driver
Osrusbf2	%wdk%\Src\Kmdf\Osrusbf2
Toaster Filter	%wdk%\Src\Kmdf\Toaster\Filter
USB Filter	%wdk%\Src\Umdf\Usb\Filter

(окончание)

Ресурсы, необходимые для данной главы	Расположение
<b>Документация WDK</b>	
PnP and Power Management in Framework-Based Drivers <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82110">http://go.microsoft.com/fwlink/?LinkId=82110</a>
Supporting PnP and Power Management in UMDF Drivers <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82112">http://go.microsoft.com/fwlink/?LinkId=82112</a>
USB Power Management <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82114">http://go.microsoft.com/fwlink/?LinkId=82114</a>
<b>Прочее</b>	
Раздел <b>Plug and Play — Architecture and Driver Support</b> <sup>4</sup> на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=82116">http://go.microsoft.com/fwlink/?LinkId=82116</a>

## Введение в Plug and Play и управление энергопотреблением

Windows и WDM предоставляют сложную модель для Plug and Play и управления энергопотреблением, которая зависит от драйвера для отслеживания, как состояния своего устройства, так и состояния системы, таким образом, в сущности, реализуя собственную неформальную машину состояний. Драйверы WDM должны знать, какие запросы Plug and Play и энергопотребления обрабатывать в каждом состоянии и какие действия предпринимать в ответ на эти запросы. Некоторые запросы требуют от драйвера выполнения одной операции, если устройство находится в нормальном режиме энергопотребления, и другой, если нет. А другие типы запросов не требуют от драйвера выполнения никаких операций, но драйвер, тем не менее, должен содержать код для анализа запросов и проверки текущего состояния системы и устройства, чтобы определить, нужно ли ему выполнять какие-либо действия.

В отличие от этой модели, инфраструктуры WDF реализуют интеллектуальное поведение по умолчанию, и предоставляют набор специфичных для определенного состояния функций обратного вызова, которые драйверы могут реализовать для нестандартной обработки событий Plug and Play и энергопотребления. WDF отслеживает состояние устройства и состояние системы, а также ведет учет информации о возможностях Plug and Play и энергопотребления драйвера и аппаратного обеспечения устройства. Инфраструктуры также управляют очередями ввода/вывода драйвера относительно состояния энергопотребления устройства. Поэтому, если при прибытии запроса ввода/вывода устройство не запитано, инфраструктура может включить его электропитание.

Драйвер WDF может подключиться к обработке более сложных ситуаций, предоставляя возможности обработки, специфические для устройства. Для этого в нем реализуются обратные вызовы функций для таких событий, как начальная инициализация, очистка ресурсов

<sup>1</sup> Plug and Play и управление энергопотреблением в драйверах, основанных на инфраструктурах. — *Пер.*

<sup>2</sup> Поддержка Plug and Play и управления энергопотреблением в драйверах UMDF. — *Пер.*

<sup>3</sup> Управление энергопотреблением устройств USB. — *Пер.*

<sup>4</sup> Plug and Play — архитектура и драйверная поддержка. — *Пер.*

после выключения устройства, подача и прекращение подачи электропитания устройству и т. п. Для обработки всех событий, для которых драйвер не реализует функций обратного вызова, применяются стандартные настройки обработки WDF.

WDF предоставляет драйверам широкий круг опций для обработки событий Plug and Play и управления энергопотреблением. Например, следующие.

- ◆ По умолчанию, как драйверы KMDF, так и драйверы UMDF поддерживают базовые возможности Plug and Play и управления энергопотреблением, включая быстрое возобновление и приостановление работы.

Драйверу необходим только код для сохранения и восстановления контекста устройства, и, для некоторых драйверов KMDF, для разрешения и запрещения прерываний устройства.

- ◆ Драйверы KMDF могут поддерживать автоматическую приостановку работы простаивающего устройства при работающей системе.

Во время периодов простоя инфраструктура переводит устройство в состояние пониженного энергопотребления. Драйверы для устройств USB могут реализовать выбороочную приостановку работы. Для этого большинству драйверов требуется вызвать лишь одну дополнительную функцию.

- ◆ Драйверы KMDF могут поддерживать подачу сигнала пробуждения для устройства и системы.

Драйвер определяет состояния энергопотребления, в которых устройство может инициировать сигнал пробуждения. Драйвер KMDF может поддерживать возможность пробуждения из любого состояния энергопотребления системы, кроме полностью выключеного. Для реализации возможности пробуждения обычно требуется добавить к драйверу всего лишь несколько функций обратного вызова.

Во всех приведенных примерах драйвер должен содержать только код, необходимый для работы со своим устройством. Функциональность для отслеживания состояния устройства и системы предоставляется инфраструктурой, которая вызывает драйвер посредством его зарегистрированных обратных вызовов для выполнения обработки, специфической для устройства.

## О механизме Plug and Play

Механизм Plug and Play представляет собой комбинацию системного программного обеспечения, аппаратного обеспечения устройства и драйверной поддержки устройства, посредством которой компьютерная система может с минимальным или вообще отсутствующим вмешательством со стороны конечного пользователя распознать изменения в конфигурации аппаратного обеспечения и настроится для работы с этими изменениями. Конечный пользователь может добавлять устройства в систему и удалять устройства из системы, не выполняя технического аппаратного конфигурирования. Например, пользователь может вставить или вытащить из разъема флэш-диск USB, или вставить портативный компьютер в док-станцию и пользоваться ее клавиатурой, мышью и монитором без необходимости выполнять вручную какие-либо дополнительные конфигурационные настройки.

Если аппаратная часть устройства и его драйвер поддерживают Plug and Play, Windows распознает новые устройства, загружает необходимые для них драйверы и запускает устройства, таким образом делая их доступными пользователю для работы. Менеджер PnP представ-

ляет собой подсистему ядра, которая распознает оборудование при первоначальной установке системы, распознает изменения в аппаратном обеспечении, внесенные между установкой и перезагрузкой системы, и реагирует на события изменения конфигурации аппаратной части при работающей системе, например на установку компьютера в док-станцию или извлечение его из нее, или вставку устройства в разъем или извлечение его из разъема.

Драйвер шины обнаруживает и перечисляет устройства и запрашивает ресурсы для этих устройств. Менеджер PnP собирает запросы на ресурсы от всех драйверов шин и назначает ресурсы этим устройствам. Ресурсы для наследуемых устройств не могут быть сконфигурированы динамически, поэтому этим устройствам менеджер PnP выделяет ресурсы в первую очередь. Если пользователь добавляет устройство, которому требуются ресурсы, которые уже были выделены, то менеджер PnP переконфигурирует выделение ресурсов.

При инициализации объекта устройства драйвер шины указывает, какие из следующих возможностей Plug and Play он поддерживает:

- ◆ выдача устройства из разъема;
- ◆ устройство является док-станцией;
- ◆ устройство можно извлечь при работающей системе;
- ◆ устройство может быть извлечено пользователем без предварительного выполнения утилиты для безопасного извлечения или выдачи устройства;
- ◆ устройство может быть заблокировано в своем разъеме, чтобы предотвратить его выброс;
- ◆ устройство можно запретить для отображения в Диспетчере устройств.

## О состояниях энергопотребления

Состояние энергопотребления описывает уровень потребления электроэнергии системой или отдельным устройством. Состояния энергопотребления системы обозначаются S0 для штатной работы, и Sx для состояний сна, где x означает номер состояния от 1 до 5. Состояния энергопотребления устройством обозначаются D0 для штатной работы, и Dx для состояний сна, где x означает номер состояния от 1 до 3. Номер состояния энергопотребления обратно пропорционален потреблению электроэнергии: в состояниях, обозначенных большими номерами, потребляется меньше электроэнергии.

В этой книге термин "состояние наивысшего энергопотребления" означает, что устройство потребляет наибольшее количество электроэнергии, а термин "состояние наименьшего энергопотребления" означает, что устройство потребляет наименьшее количество электроэнергии. Таким образом, состояние энергопотребления D0 более высокое, чем состояние D1, а состояние D3 — более низкое, чем D2.

Для системы S0 является наивысшим состоянием энергопотребления, в котором система находится, когда полностью используются все ее функциональные возможности.

В состоянии S4 система находится в спящем режиме<sup>1</sup> или гибернации (hibernation).

В состоянии S5 система выключена.

---

<sup>1</sup> Спящий режим — состояние, в котором компьютер завершает работу после сохранения всего содержимого памяти на жестком диске. При выводе компьютера из спящего режима все открытые до этого приложения и документы восстанавливают свое состояние. — Пер.

В состояниях S1, S2 и S3 система находится в ждущем режиме<sup>1</sup> (sleep mode), с соответственно последовательно более низкими уровнями потребления электроэнергии.

Для устройств D0 является состоянием штатной работы при полной запитке. В состоянии D3 устройство обесточено (выключено). Все устройства должны поддерживать эти два состояния. В отличие от состояний энергопотребления системы, где состояния S4 и S5 определены универсально, точные определения для промежуточных состояний энергопотребления устройств зависят от конкретной шины и устройства. Не все устройства поддерживают промежуточные состояния энергопотребления, и многие поддерживают только состояния D0 и D3.

### Примечание

Для устройств PCI спецификация PCI определяет состояния D3hot и D3cold. В Windows состояние D3hot означает, что устройство находится в состоянии D3, а его родительская шина — в состоянии D0. А состояние D3cold означает, что устройство находится в состоянии D3, а его родительская шина — в состоянии Dx.

Устройство может переходить из состояния D0 в любое более низкое состояние (Dx) энергопотребления, и с любого состояния пониженного энергопотребления в состояние D0. Но устройство не может переходить непосредственно из одного состояния пониженного энергопотребления (Dx) в другое; для этого ему нужно сначала возвратиться в состояние D0. Кроме этого, устройство должно быть в состоянии D0, когда драйвер разрешает или запрещает сигнал пробуждения. Причиной этому является тот факт, что доступ к аппаратной части устройства запрещен, когда устройство находится в ждущем режиме; поэтому, прежде чем осуществлять какую-либо деятельность с аппаратной частью, драйвер должен возвратить устройство в рабочее состояние.

Состояние энергопотребления устройства связано с состоянием энергопотребления системы, но не обязательно должно быть таким же. Например, многие устройства могут находиться в выключенном состоянии (D3), когда система находится в рабочем состоянии (S0). Обычно состояние энергопотребления устройства не может быть выше состояния энергопотребления системы, т. к. многие устройства берут электропитание от системы. Устройства, которые могут выводить систему из состояния сна, — скорее исключение, нежели правило. Такие устройства обычно находятся в ждущем состоянии (D1 или D2), когда система находится в ждущем состоянии.

**UMDF и возможности энергопотребления устройства.** Для драйверов UMDF, возможности энергопотребления устройства устанавливаются основным драйвером шины, и драйвер UMDF не может изменить эту настройку.

**KMDF и возможности энергопотребления устройства.** Драйвер KMDF может установить возможности энергопотребления устройства таким же образом, как он устанавливает возможности Plug and Play. Обычно драйвер шины устанавливает возможности устройств, подключенных к его шине, но функциональный драйвер или драйвер фильтра может заменить настройки, выполненные драйвером шины. Это следующие настройки:

- ◆ состояния энергопотребления, поддерживаемые устройством в дополнение к состояниям D0 и D3;

<sup>1</sup> Ждущий режим — состояние, в котором компьютер при простое потребляет меньше электроэнергии, но остается доступным для немедленного применения. При этом данные из памяти не записываются на жесткий диск. При прерывании питания данные из памяти будут потеряны. — Пер.

- ◆ состояния энергопотребления, из которых устройство может реагировать на сигнал пробуждения;
- ◆ наивысшее состояние энергопотребления Dx, поддерживаемое устройством для каждого состояния Sx системы;
- ◆ наивысшее состояния энергопотребления устройства Dx, из которого устройство может инициировать сигнал пробуждения системы;
- ◆ наимизшее состояние энергопотребления системы Sx, при котором устройство может инициировать сигнал пробуждения;
- ◆ приблизительное время задержки при возвращении устройства из каждого состояния Dx в состояние D0;
- ◆ "идеальное" состояние Dx, в которое должно перейти устройство, когда его сигнал пробуждения не разрешен, а система переходит в ждущий режим.

Когда система переходит в состояние Sx, инфраструктура переводит устройство или в идеальное состояние Dx, или в наимизшее состояние Dx, которое устройство может поддерживать при данном состоянии Sx, какое из них выше.

Драйвер докладывает о возможностях энергопотребления инфраструктуре, а инфраструктура, в свою очередь, информирует о них систему. Драйверы в стеке устройств устанавливают возможности энергопотребления на кооперативных началах, поэтому другой, более высший в стеке, драйвер может изменить значения, установленные драйвером KMDF.

## Политика энергопотребления

Политика энергопотребления устройства определяет состояние энергопотребления, в котором устройство должно находиться в любой момент времени. В каждом стеке устройства ответственным за управление политикой энергопотребления устройства является один драйвер, называемый владельцем политики энергопотребления для стека устройства:

- ◆ драйвер UMDF должен указывать явно, что он управляет политикой энергопотребления;
- ◆ по умолчанию KMDF полагает, что владельцем политики энергопотребления устройства является объект FDO. Если устройство контролируется сырьим объектом PDO, KMDF полагает, что владельцем политики энергопотребления устройства является данный "сырой" объект PDO.

Ответственным за функциональную работу устройства является функциональный драйвер, поэтому, скорее всего, именно он имеет необходимую информацию о том, как наилучшим образом управлять энергопотреблением устройства. Драйвер фильтра KMDF может указывать, что он является владельцем политики энергопотребления для своего устройства, извещая об этом инфраструктуру во время инициализации.

Владельцем политики энергопотребления не обязательно является тот драйвер, который манипулирует аппаратной частью устройства, чтобы изменить состояние энергопотребления. Это просто драйвер, который указывает, когда устройство должно переходить в определенное состояние энергопотребления.

Драйвер, заявляющий право на владение политикой энергопотребления, должен с помощью какого-либо механизма, не входящего в рамки стандартного управления операционной системы, удостоверится в том, что он в самом деле является единственным владельцем политики энергопотребления в своем стеке. Обычно таким механизмом является задокументиро-

ванная политика, указывающая, какой драйвер является владельцем политики энергопотребления.

## Plug and Play и управление энергопотреблением в WDF

WDF реализует возможности Plug and Play и управления энергопотреблением с помощью нескольких внутренних машин состояния. Как KMDF, так и UMDF используют одни и те же машины состояния. Событие ассоциируется с конкретными действиями, которые драйверу может быть нужно будет выполнить в определенное время, а драйвер реализует обратные вызовы функций для события, для выполнения действий, требующиеся его устройству. Функции обратного вызова вызываются в определенном порядке, и каждая из них подчиняется "контракту", для того чтобы гарантировать, что и устройство, и система будут находиться в определенном состоянии, когда драйвер вызывается для выполнения действия.

### Стандартные настройки Plug and Play и управления энергопотреблением

Хотя WDF предоставляет драйверам большую гибкость в управлении подробными аспектами возможностей Plug and Play своего устройства, в ней также реализованы функциональные возможности по умолчанию, которые позволяют многим драйверам фильтра и драйверам исключительно программных устройств<sup>1</sup> совсем не содержать кода для работы с Plug and Play. По умолчанию WDF поддерживает все возможности Plug and Play, в которых такие драйверы могут нуждаться.

По умолчанию WDF полагает следующее:

- ◆ устройство поддерживает состояния D0 и D3;
- ◆ устройство и драйвер не поддерживают бездействие<sup>2</sup> и пробуждение;
- ◆ для очередей ввода/вывода объектов FDO и PDO применяется управление энергопотреблением.

#### Машины состояний и четкий контракт

История машин состояний в инфраструктуре позволяет увидеть, насколько мы недооценивали, как сложно будет их реализовать. Когда Джейк Ошинс (Jake Oshins) впервые подал свою идею применения формализованных машин состояний, применив диаграммы UML<sup>3</sup> для визуального представления этой концепции, существовало две машины состояний (PnP и энергопотребления), для которых было меньше десяти состояний. Надо сказать, для энергопотребления было всего лишь два состояния — включено и выключено!

Не нужно и говорить, что количество состояний увеличилось драматически, почти до 300 состояний для всех машин состояний. Возросло не только количество состояний, но

<sup>1</sup> В дальнейшем, во избежание многословности, такие драйверы будем называть "чисто программными драйверами". — Пер.

<sup>2</sup> Под бездействием здесь и дальше подразумевается, что устройство не выполняет никаких операций, и поэтому было переведено в одно из состояний пониженного энергопотребления. — Пер.

<sup>3</sup> Стандартный язык для наглядного описания и представления различных частей систем программного обеспечения при его разработке. — Пер.

также и количество машин состояний. Кроме машин состояний Plug and Play, энергопотребления и политики энергопотребления, в инфраструктуре также используются машины состояний для управления логикой бездействия и обратных вызовов функций для самоуправляемого ввода/вывода.

Имея сейчас перед собой конечную реализацию, приступая к работе в то время у нас не было ни малейшего понятия о том, какую громадную работу нам предстояло проделать, и мы никогда не могли бы предвидеть конечный результат. Самое большее удовольствие, которое я получил от работы над машинами состояний, было то, что нам удалось оставить всю сложность внутри инфраструктуры. Все, с чем приходится работать разработчику драйвера, — это четкий контракт с очень ясными указаниями, и ему никогда не нужно разбираться со всеми подробностями самому. (*Даун Холэн (Down Holan), команда разработчиков Windows Driver Foundation, Microsoft.*)

## **Очереди ввода/вывода и управление энергопотреблением**

Инфраструктуры реализуют управление энергопотреблением для очередей ввода/вывода, чтобы очередь автоматически запускалась и останавливалась, когда устройство переходит в рабочий режим и выходит из него соответственно. О такой очереди говорят, что она управляется энергопотреблением (power managed). Инфраструктура посылает драйверу запросы ввода/вывода из управляемой энергопотреблением очереди только тогда, когда устройство доступно и находится в рабочем состоянии энергопотребления. Драйверу не требуется сдерживать информацию о состоянии устройства или проверять состояние устройства каждый раз, когда он получает запрос ввода/вывода от управляемой энергопотреблением очереди.

По умолчанию все очереди ввода объектов FDO и PDO управляются энергопотреблением. Драйвер может с легкостью изменить эти настройки по умолчанию, чтобы можно было создавать очереди, неуправляемые энергопотреблением, или чтобы можно было сконфигурировать управляемые энергопотреблением очереди для объекта FiDO. Если при прибытии запроса ввода/вывода устройство бездействует в состоянии низкого энергопотребления, инфраструктура может восстановить его энергопотребление до рабочего, прежде чем доставлять запрос драйверу.

## **Обратные вызовы функций для событий Plug and Play и управления энергопотреблением**

Большинство обратных вызовов функций для Plug and Play и энергопотребления определено попарно: один вызов для события, происходящего при переходе в состояние, а второй — для события, происходящего при выходе из состояния. Обычно, один из членов пары выполняет задачу обратную выполняемой вторым членом пары. Драйвер может реализовывать один, оба или ни одного из вызовов. Драйвер UMDF, в котором обе функции определены на одном интерфейсе, должен реализовать весь интерфейс на объекте обратного вызова устройства; при этом необязательные функции можно реализовать лишь в минимальном исполнении.

Инфраструктуры рассчитаны работать совместно с драйверами, заявившими о своем желании принимать участие в обработке запросов. Драйвер реализует обратные вызовы функций для обработки только тех событий, которые затрагивают его устройство. Например, некоторые драйверы должны сохранять состояние устройства непосредственно перед выходом устройства из состояния энергопотребления D0 и восстанавливать состояние устройства после возвращения устройства в состояние энергопотребления D0. Или же, устройство мо-

жет быть оборудовано мотором или вентилятором, который драйвер должен включить при переходе устройства в состояние D0 и выключить при выходе устройства из состояния D0. Для выполнения таких задач драйвер может реализовать функции обратного вызова, которые инициируются этими событиями. Если же устройству не требуется обслуживание в такие моменты, то его драйвер и не предоставляет такового, т. е. не реализует функций обратного вызова.

В табл. 7.1 перечислены типы возможностей Plug and Play и энергопотребления, которые могут требоваться драйверу, интерфейсы UMDF и событийные функции обратных вызовов KMDF, которые драйвер реализует для поддержки этих возможностей.

**Таблица 7.1. Обратные вызовы функций драйверов WDF для событий Plug and Play и энергопотребления**

Если драйвер	Необходимо реализовать следующий интерфейс UMDF и его методы на объекте обратного вызова устройства	Необходимо реализовать следующую событийную функцию обратного вызова KMDF
Использует самоуправляемый ввод/вывод	<code>IPnpCallbackSelfManagedIo:::Xxx</code>	<code>EvtDeviceSelfManagedIoXxx</code>
Требует обслуживания непосредственно перед первоначальным подключением питания устройства и после отключения для перераспределения ресурсов или удаления устройства	<code>IPnpCallbackHardware:::OnPrepareHardware</code> <code>IPnpCallbackHardware:::OnReleaseHardware</code>	<code>EvtDevicePrepareHardware</code> и <code>EvtDeviceReleaseHardware</code>
Требует обслуживание сразу же после перехода устройства в состояние D0 и непосредственно перед выходом из состояния D0	<code>IPnpCallback:::OnDOEntry</code> <code>IPnpCallback:::OnDOExit</code>	<code>EvtDeviceDOEntry</code> и <code>EvtDeviceDOExit</code>
Требует возможности для оценки и возможного запрета каждой попытки остановить или удалить устройство	<code>IPnpCallback:::OnQueryStop</code> <code>IPnpCallback:::OnQueryRemove</code>	<code>EvtDeviceQueryStop</code> и <code>EvtDeviceQueryRemove</code>
Требует дополнительного обслуживания при неожиданном удалении, выходящего за рамки обычной обработки при удалении устройства	<code>IPnpCallback:::OnSurpriseRemoval</code>	<code>EvtDeviceSurpriseRemoval</code>

Так как драйверы KMDF имеют большую свободу доступа к аппаратному обеспечению устройства, чем драйверы UMDF, инфраструктура KMDF поддерживает дополнительные возможности, например, пробуждение системы. В табл. 7.2 приведены дополнительные функции обратного вызова, применимые только с драйверами KMDF.

**Таблица 7.2. Дополнительные функции обратного вызова KMDF для событий Plug and Play**

Если драйвер	Необходимо реализовать следующую событийную функцию обратного вызова KMDF
Управляет ресурсными требованиями устройства	<code>EvtDeviceResourceRequirementsQuery</code> <code>EvtDeviceResourcesQuery</code> <code>EvtDeviceRemoveAddedResources</code> <code>EvtDeviceFilterAddResourceRequirements</code> <code>EvtDeviceFilterRemoveResourceRequirements</code>

Таблица 7.2 (окончание)

Если драйвер	Необходимо реализовать следующую событийную функцию обратного вызова KMDF
Управляет сигналом пробуждения устройства	<i>EvtDeviceArmWakeFromSx</i> и <i>EvtDeviceDisarmWakeFromSx</i>  <i>EvtDeviceArmWakeFromS0</i> и <i>EvtDeviceDisarmWakeFromS0</i>  <i>EvtDeviceEnableWakeAtBus</i> и <i>EvtDeviceDisableWakeAtBus</i>  <i>EvtDeviceWakeFromSxTriggered</i> и <i>EvtDeviceWakeFromS0Triggered</i>
Работает с аппаратным обеспечением, выполняя задачи, связанные с прерываниями	<i>EvtInterruptEnable</i> и <i>EvtInterruptDisable</i>  <i>EvtDeviceD0EntryPostInterruptsEnabled</i> и <i>EvtDeviceDExitPreInterruptsDisabled</i>

Если вы знакомы с драйверами WDM, то, наверное, помните, что при любом изменении состояния энергопотребления системы владелец политики энергопотребления WDM должен определить необходимое состояние энергопотребления для своего устройства, а потом послать запросы управления энергопотреблением, чтобы перевести свое устройство в это состояние в соответствующее время. Машина состояний WDF автоматически преобразует системные события энергопотребления в события энергопотребления устройства и извещает драйвер о необходимости выполнить следующие действия:

- ◆ перевести устройство в режим низкого энергопотребления, когда система переходит в состояние Sx;
- ◆ возвратить устройство обратно в режим полного энергопотребления, когда система возвращается в состояние S0;
- ◆ разрешить сигнал пробуждения устройства, чтобы он мог быть активирован, когда устройство находится в состоянии Dx, а система — в рабочем состоянии (применимо только к KMDF);
- ◆ разрешить сигнал пробуждения устройства, чтобы он мог быть активирован, когда устройство находится в состоянии Dx, а система — в режиме ожидания (применимо только к KMDF).

KMDF автоматически поддерживает правильное поведение во взаимоотношениях между родителями и дочерними устройствами для драйверов шины. Если родительское и дочернее устройства обесточены, то KMDF обеспечивает, что перед тем, как родительское устройство может перевести дочернее устройство в состояние D0, оно само получает питание.

## Поддержка бездействия и пробуждения (только для KMDF)

Для управления бездействующими устройствами, инфраструктура извещает драйвер о необходимости перевести устройство из рабочего состояния в необходимое состояние низкого энергопотребления, когда устройство не выполняет никаких операций, и возвратить устройство в рабочее состояние, когда необходимо обработать запросы. Драйвер предоставляет функции обратного вызова, которые инициализируют и деинициализируют устройство, со-

храняют и восстанавливают состояние устройства и разрешают и запрещают сигнал пробуждения устройства.

По умолчанию пользователь, имеющий необходимые привилегии, может управлять как поведением бездействующего устройства, так и способностью устройства пробуждать систему. KMDF реализует необходимого поставщика WMI, а Диспетчер устройств выводит на экран страницу свойств для установки необходимой конфигурации. Владелец политики энергопотребления устройства может отключить эту возможность, указывая соответствующее значение перечисления при выполнении инициализации определенных настроек политики энергопотребления.

## Энергостраничные и неэнергостраничные драйверы

Большинство устройств можно перевести в режим пониженного энергопотребления, не затрагивая способность системы обращаться к файлу подкачки или записывать файл гибернации. Драйверы для таких устройств считаются энергостраничными (power pageable):

- ◆ все драйверы UMDF являются энергостраничными;
- ◆ большинство драйверов KMDF являются энергостраничными.

Но устройство, которое находится в пути гибернации<sup>1</sup> (hibernation path), должно оставаться в состоянии D0 при некоторых переходах из одного состояния энергопотребления в другое, чтобы система могла записать и сохранить файл гибернации. Устройство, находящееся в пути подкачки (paging path)<sup>2</sup>, должно оставаться в состоянии D0 до тех пор, пока система не запишет файл гибернации, после чего вся компьютерная система отключается от питания. Стеки устройств для устройств гибернации и подкачки принято называть неэнергостраничными (non-power pageable). Драйвер KMDF указывает, что он может поддерживать файл подкачки, гибернации или системного дампа, вызывая метод `WdfDeviceSetSpecialFileSupport` и предоставляя обратный вызов для извещения, если устройство в действительности используется для таких файлов.

Например, драйверы в стеках видеоустройств и устройства хранения являются неэнергостраничными, т. к. система использует эти устройства при выключении питания системы. Монитор должен оставаться включенным, чтобы Windows могла выводить информацию для пользователя. При переходе системы в состояние S4 целевой диск для файла гибернации и диск, на котором находится файл подкачки, должны оставаться в состоянии D0, чтобы система могла записать файл гибернации. Чтобы эти диски могли продолжать получать питание, все устройства, от которых они зависят (например, контроллер диска, шина PCI, контроллер прерываний и т. д.), также должны продолжать получать питание. Поэтому драйверы в стеках всех этих устройств должны быть неэнергостраничными.

Большинство драйверов должны использовать стандартные настройки инфраструктуры.

- ◆ По умолчанию объекты FDO являются энергостраничными.
- ◆ По умолчанию объекты PDO наследуют настройки драйвера, перечислившего их.

Если объект PDO является энергостраничным, то все объекты устройств, подключенные к нему, также должны быть энергостраничными. По этой причине, драйвер шины обыч-

<sup>1</sup> Путь гибернации содержит устройство, на котором система записывает файл гибернации, а также любые другие устройства вдоль пути от корня к этому устройству, которые необходимы для обеспечения подачи питания этому устройству. — *Пер.*

<sup>2</sup> Устройство находится в пути подкачки, если оно принимает участие в операциях ввода/вывода с файлом подкачки. — *Пер.*

но обозначает свой объект FDO как неэнергостраничный, чтобы его объект PDO унаследовал такой же атрибут. Тогда объекты устройств, которые загружаются поверх объекта PDO, могут быть или энергостраничными, или неэнергостраничными.

- ◆ Объект FiDO использует те же настройки, что и драйвер, находящийся в стеке непосредственно под ним. Драйвер не может изменить настройки объекта FiDO.

Если настройки по умолчанию не подходят, драйвер шины или функциональный драйвер может изменить их, явно вызвав метод `WdfDeviceInitSetPowerPageable` или метод `WdfDeviceInitSetPowerNotPageable` во время инициализации объекта. Эти методы устанавливают и сбрасывают значение `DO_POWER_PAGABLE` в поле `Flags` в базовом объекте устройства WDM для объектов FDO или PDO, но не влияют на объекты FiDO.

Инфраструктура может изменить значение флага `DO_POWER_PAGABLE` для любого объекта устройства, если система известит драйвер о том, что устройство используется для хранения файла подкачки, гибернации или дампа.

Если вы уверены в том, что в стеке устройств нет ни одного драйвера, который должен быть неэнергостраничным, то драйвер может вызывать метод `WdfDeviceInitSetPowerPageable`. Такая уверенность может быть в том случае, если все драйверы в стеке были созданы вами, или если требования стека устройства четко и ясно задокументированы. Объект PDO может быть энергостраничным только в том случае, если стеки устройств всех дочерних устройств также являются энергостраничными.

KMDF предоставляет следующие возможности специальной обработки для обеспечивающих подкачку драйверов.

- ◆ Инфраструктура запрещает, но не отключает, прерывание устройства, когда устройство покидает состояние D0. Инфраструктура не может отключить прерывание, т. к. требуемая для этого системная функция `IoDisconnectInterruptXxx` содержит страничный код.
- ◆ Инфраструктура реализует таймер WDT (WatchDog Timer, программируемый сторожевой (контрольный) таймер, "будильник") во всех обратных вызовах для событий энергопотребления и пробуждения. Если драйвер вызывает обращения ввода/вывода к файлу подкачки после того, как устройство хранения файла подкачки вышло из состояния D0, система зависает, т. к. отсутствует устройство для обслуживания запроса. По истечению отсчета таймера WDT происходит сбой системы, чтобы пользователь мог выяснить, какой из драйверов вызвал взаимоблокировку. При отладке драйвера длительность тайм-аута таймера WDT можно продлить с помощью расширения отладчика `!wdfextendwatchdog`. KMDF не предоставляет возможности увеличить значение тайм-аута таймера WDT программным способом.
- ◆ Драйвер может определить, находится ли он в данное время в неподдерживающем подкачку состоянии энергопотребления (non-pageable power state<sup>1</sup>), вызвав метод `WdfDevStateIsNP(WdfDeviceGetDevicePowerState())` из функции обратного вызова для события энергопотребления или политики энергопотребления.

Метод `WdfDeviceGetDevicePowerState` возвращает значение перечисления `WDF_DEVICE_POWER_STATE`, указывающее подробное состояние машины состояний инфра-

---

<sup>1</sup> Если машина состояний инфраструктуры находится в не поддерживаемом подкачу состоянию, то драйвер не поддерживает подкачу и не должен выполнять никаких операций, которые могут вызвать обращение операционной системы к файлу подкачки. Такие операции включают обращение к файлам, реестру или страничному пулу. — Пер.

структуры. Например, значения WdfDevStatePowerD0 и WdfDevStatePowerD0NP представляют поддерживающее подкачку (pageable power state) состояние D0 и не поддерживающее подкачку состояние D0 соответственно.

Метод wdfDevStateIsNP возвращает TRUE, если в настоящее время драйвер находится в не-поддерживающем подкачку состоянии энергопотребления, и FALSE — в противном случае. Это значение действительно только на протяжении текущего исполнения функции обратного вызова. Состояние энергопотребления может измениться после возвращения функцией управления. Поэтому, если драйвер должен выполнять действия, требующие использования файла подкачки, он должен делать это сразу же после определения, что состояние энергопотребления устройства разрешает такие действия.

## Порядок обратных вызовов функций для событий Plug and Play и управление энергопотреблением

Механизм Plug and Play и управление энергопотреблением осуществляют все операции, требуемые для приведения вставленного в разъем устройства в полностью рабочее состояние и для удаления работающего устройства из системы. Когда пользователь вставляет в разъем новое устройство, система должна определить тип и возможности этого устройства, назначить ему ресурсы, работать совместно с драйверами устройства, чтобы запитать и инициализировать его, а также выполнить любые другие операции, необходимые для приведения устройства в рабочее состояние. Когда пользователь вынимает устройство из разъема, система выполняет обратную последовательность действий. Основные операции при подключении и запуске устройства следуют в установленном порядке, как и основные операции при остановке и удалении устройства.

На каждом этапе этой последовательности стек устройств находится в четко определенном состоянии. Машина состояний WDF отслеживает состояние стека устройств при переходах из одного состояния в другое, и WDF определяет функции обратного вызова для многих из этих переходов. Когда происходит изменение состояния, инфраструктура вызывает функцию обратного вызова (если она имеется), которая соответствует новому состоянию.

Например, одной из ключевых операций в цикле запуска является инициализация устройства и драйвера при включении питания устройства. В большинстве стеков устройств ответственным за обеспечение питания устройства является драйвер шины, а за инициализацию — функциональный драйвер. В зависимости от типа устройства, драйвер может выполнять инициализацию до подачи питания, после или как до, так и после. WDF определяет обратные вызовы для всех этих состояний. Если устройство генерирует прерывание, драйвер может активировать обратный вызов после включения питания, но до подключения прерывания, или же немедленно после подключения прерывания.

Состояние стека устройств может измениться по разным причинам, самыми распространеными из которых являются следующие:

- ◆ добавление устройства в систему;
- ◆ удаление устройства из системы, или должным образом, или неожиданно;
- ◆ включение питания системы;
- ◆ система находится в спящем или ждущем режиме;

- ◆ выключение питания системы;
- ◆ пристаивающее устройство переводится в состояние пониженного энергопотребления;
- ◆ устройство переводится из состояния пониженного энергопотребления в рабочее состояние для обслуживания прибывшего запроса ввода/вывода или по внешнему сигналу пробуждения;
- ◆ добавление другого устройства в систему, что требует перераспределения ресурсов операционной системой;
- ◆ обновление или переустановка драйвера.

Какие именно обратные вызовы инфраструктура задействует, зависит от природы изменения состояния. Например, при выключении питания системы инфраструктура вызывает все функции обратного вызова драйвера, вовлеченные в остановку всех функций ввода/вывода устройства и выключения питания устройства. Но при выключении питания пристаивающего устройства требуется задействовать только обратные вызовы для остановки только некоторых функций ввода/вывода устройства и для выключения питания устройства. Инфраструктура удерживает объект устройства и его ресурсы для применения, когда устройство возобновит нормальную работу.

Но, с точки зрения драйвера, причина изменения состояния не представляет важности. Функции обратного вызова драйвера выполняют дискретные задачи в четко определенное время, например, инициализацию устройства после включения питания его аппаратной части. Но для драйвера не имеет значения причина, по какой устройству включается питание, будь то в результате первоначальной установки устройства или в результате перехода в рабочее состояние по сигналу пробуждения. Для драйвера важным является то, что инфраструктура всегда активирует функции обратного вызова в одном и том же порядке и в соответствии с контрактом для данного состояния.

**Изменения состояний и последовательность обратных вызовов.** Инфраструктура активирует релевантные обратные вызовы в установленном порядке при вставлении устройства в разъем и приведении его в полностью рабочее состояние, а также в обратном порядке при отключении питания и извлечении устройства. Но устройства вставляются и извлекаются не так уж и часто. Более типичными являются ситуации с переходом системы в состояние Sx, когда пользователь закрывает крышку ноутбука, или с отключением драйвером питания сетевой платы, когда ее сетевой кабель извлечен из разъема. В таких ситуациях выполнять всю последовательность обратных вызовов нет необходимости. Вместо этого инфраструктура вызывает только те функции последовательности, которые являются необходимыми для перевода устройства в требуемое состояние.

Во время отключения питания инфраструктура останавливает исполнение последовательности обратных вызовов в одной из четырех точек, в зависимости от состояния устройства:

- ◆ устройство переходит в режим более низкого энергопотребления;
- ◆ устройство останавливается для перераспределения ресурсов;
- ◆ устройство запрещено или удалено, но присутствует физически, например, при отключении устройства пользователем в Диспетчере устройств;
- ◆ устройство физически удалено.

После выполнения каждого последующего обратного вызова, и в каждой следующей точке остановки выполнения последовательности обратных вызовов, устройство и его драйвер

предоставляют все меньше функциональных возможностей. Когда инфраструктура возвращает устройство в рабочее состояние, она начинает последовательность обратных вызовов в той же самой точке, в которой она остановилась, но в обратном порядке. При неожиданном извлечении устройства последовательность обратных вызовов несколько иная и описывается в разд. *"Неожиданное извлечение"* далее в этой главе.

После выхода устройства из рабочего состояния и перехода в состояние пониженного энергопотребления, перед следующим изменением его состояния энергопотребления инфраструктура всегда возвращает устройство в рабочее состояние. Например, допустим, что устройство существует в состоянии D2, когда пользователь выключает систему. Инфраструктура начинает процесс выключения с точки остановки и вызывает функции обратного вызова в обратной последовательности, чтобы возвратить устройство в состояние D0. После этого она запускает последовательность снова, начиная с устройством в рабочем состоянии и выполняя необходимые действия для его выключения. Хотя такое поведение может казаться идущим вразрез с интуицией, этот подход обеспечивает, что драйверы устройства могут выполнить любые дополнительные операции, требуемые при выключении системы. Информация состояния, сохраненная драйвером при переходе в состояние D2, может быть недостаточной для состояния D3, поэтому промежуточный переход в состояние D0 позволяет драйверу получить всю дополнительную информацию, необходимую для перехода в состояние D3.

**Последовательности запуска и остановки.** Чтобы обеспечить реализацию драйвером соответствующих функций обратных вызовов и выполнение каждой из этих функций надлежащих задач, необходимо понимать последовательность, в которой инфраструктура вызывает эти функции.

На рис. 7.1 показаны последовательности действий при запуске и остановке устройства. Действия при запуске в левом столбце соответствуют действиям при остановке в правом. Не все действия применимы к каждому типу драйвера или объекту устройства; применимость действия зависит от причины, по которой выполняется запуск или остановка. Например, инфраструктура не вызывает драйвер для разрешения сигнала пробуждения, когда устройство останавливается для перераспределения системных ресурсов менеджером PnP. А неожиданное извлечение устройства активирует последовательность выключения и удаления, хотя оборудование уже было физически удалено.

Конкретные функции обратного вызова, которые инфраструктура активирует для каждого из этих действий, приводятся далее в этой главе.

**Неуспешные завершения функций обратного вызова.** Если в любом из шагов последовательности функция обратного вызова возвращает статус неудачного выполнения, инфраструктура удаляет стек устройства.

Если сбой происходит во время включения устройства, инфраструктура вызывает функцию обратного вызова для освобождения аппаратной части (если драйвер реализует таковую), но не вызывает никаких других функций обратного вызова.

Если сбой происходит во время выключения устройства, инфраструктура продолжает вызывать функции обратного вызова драйвера. Поэтому функции обратного вызова в последовательности выключения устройства должны выдерживать сбои, вызванные не реагирующими оборудованием.

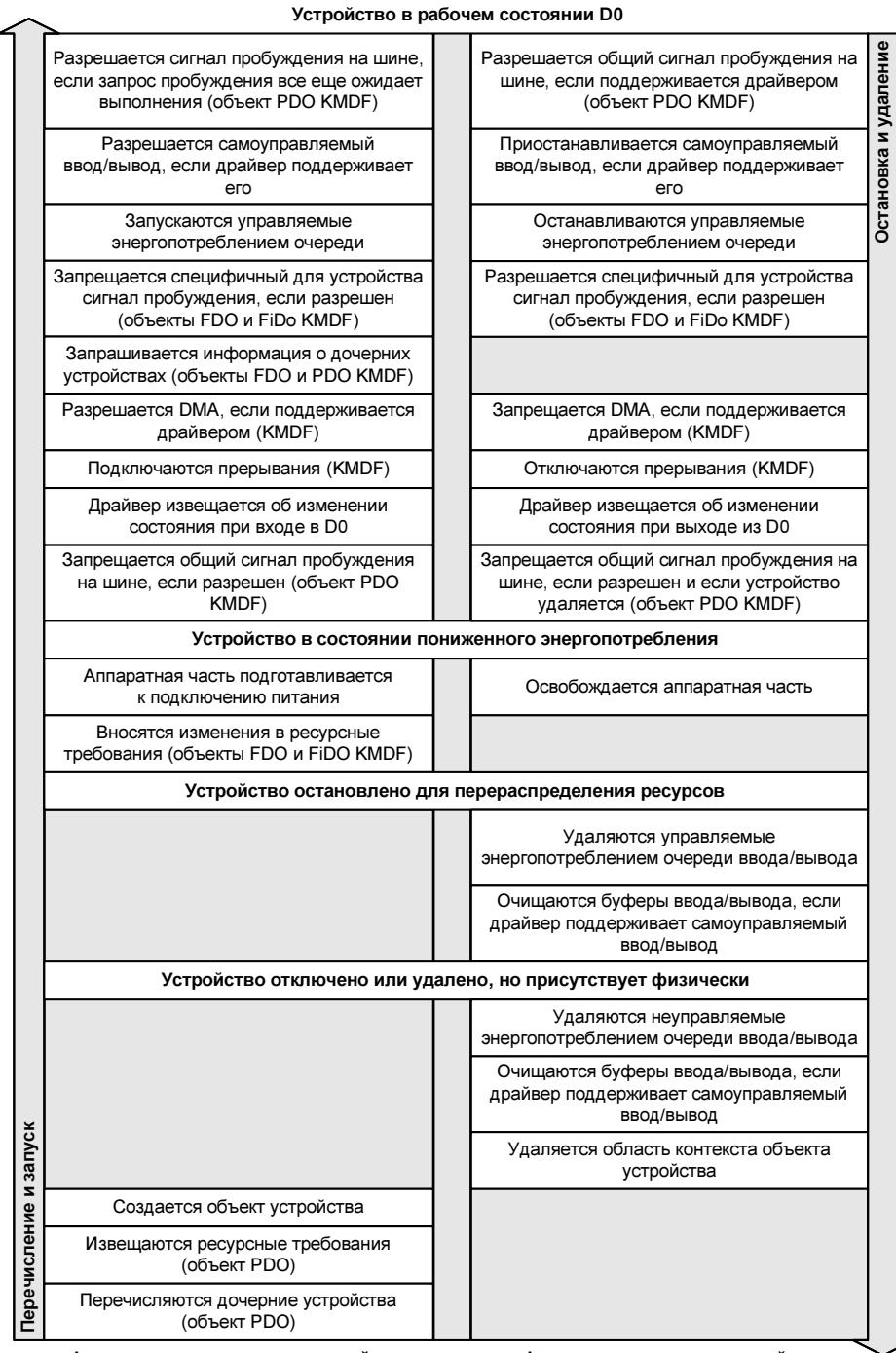


Рис. 7.1. Последовательности действий при запуске и остановке устройства инфраструктурой

## Перечисление и запуск устройств

Всякий раз при запуске Windows загрузчик, менеджер PnP и драйверы совместно перечисляют все устройства в системе и создают узлы devnode для их представления. То же самый процесс, только несколько уменьшенного масштаба, происходит при добавлении устройства к работающей системе.

Во время перечисления устройств менеджер PnP загружает драйверы и создает стек устройств для каждого устройства, подключенного к системе. Чтобы подготовить устройство к работе, менеджер PnP посыпает последовательность запросов каждому стеку устройств, чтобы получить информацию о требуемых устройству ресурсах, его возможностях и т. п. Драйверы в стеке устройств отвечают на эти запросы по очереди, начиная с низу стека, с наиболее близкого к аппаратной части драйвера. Таким образом, объект PDO включает питание устройства, перед тем, как объект FiDO получает запрос на выполнение своих обязанностей по включению устройства. Отключение устройства происходит в обратном порядке. В общем драйверы, расположенные выше в стеке, зависят от функциональности драйверов, расположенных в стеке ниже них, поэтому они начинают работу позже и заканчивают раньше.

Инфраструктуры принимают участие в этой последовательности от имени своих драйверов. Инфраструктуры реагируют немедленно на запросы менеджера PnP, когда они могут и вызывают функции обратного вызова драйвера для предоставления информации и выполнения задач, для которых драйверу требуется выполнить ввод или действия. Для драйверов инфраструктуры.

- ◆ Все драйверы UMDF могут реализовать один и тот же набор интерфейсов обратного вызова.

Интерфейс `IDriverEntry` реализуется на объекте обратного вызова драйвера. Интерфейсы `IPnpCallback`, `IPnpCallbackHardware` и `IPnpSelfManagedIo` реализуются на объекте обратного вызова устройства. Интерфейс `IQueueCallbackXXX` реализуется на объекте обратного вызова очереди.

- ◆ Для драйверов KMDF большинство функций обратного вызова для событий применимы к любому типу объектов устройств.

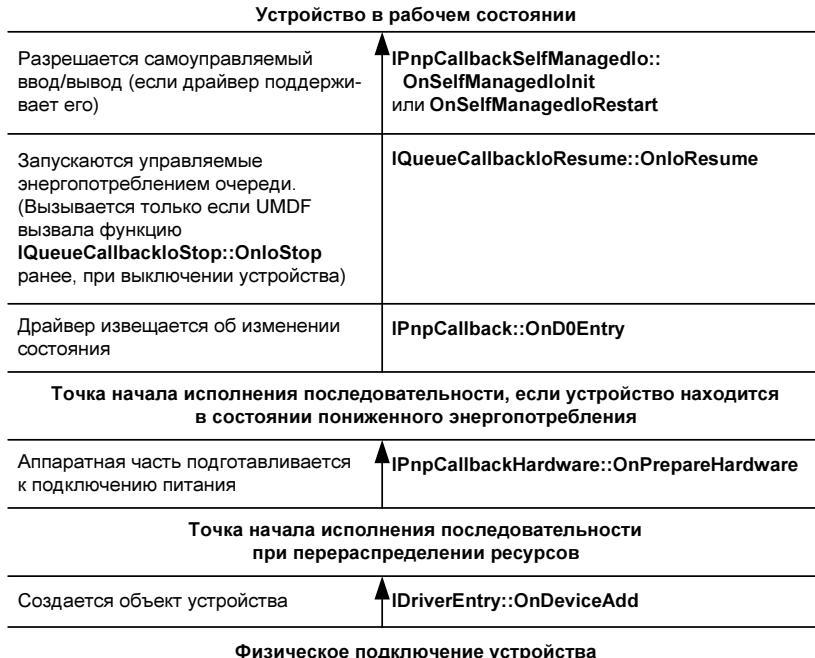
Но инфраструктура вызывает некоторые функции обратного вызова только для объектов PDO, а другие — только для объектов FDO и FiDO.

На рис. 7.2 и 7.3 показаны последовательности вызовов функций обратного вызова для драйверов UMDF и для объектов FDO и FiDO KMDF, которые принимают участие в приведении устройства в полностью рабочее состояние, начиная с состояния "Физическое подключение устройства" внизу рисунков.

На этих рисунках горизонтальные линии обозначают шаги, выполняемые при запуске устройства. В левом столбце рисунка описывается шаг, а в правом — методы обратного вызова, выполняющие этот шаг. Если устройство останавливается для того, чтобы позволить менеджеру PnP перераспределить системные ресурсы или если устройство присутствует физически, но находится в нерабочем состоянии, выполнению подлежат не все шаги, как можно видеть в рисунках.

Вне нижнего предела рис. 7.2 устройство физически не присутствует в системе. Когда пользователь физически вставляет устройство в разъем, инфраструктура начинает выполнять последовательность, вызывая метод обратного вызова `IDriverEntry::OnDeviceAdd`, чтобы

драйвер мог создать объект обратного вызова устройства и инфраструктурный объект устройства для представления устройства. Инфраструктура продолжает вызывать функции обратного вызова драйвера, продвигаясь вверх по последовательности до тех пор, пока устройство не будет приведено в состояние полной рабочей готовности.



**Рис. 7.2.** Последовательность шагов при перечислении и запуске устройства для драйвера UMDF

На рис. 7.3 показана последовательность обратных вызовов функций для объектов FDO и FIDO KMDF, которые принимают участие в приведении устройства в полностью рабочее состояние.

Вне нижнего предела рис. 7.3 устройство физически не присутствует в системе. Когда пользователь физически вставляет устройство в разъем, инфраструктура KMDF начинает последовательность, вызывая функцию обратного вызова *EvtDriverDeviceAdd*, чтобы драйвер мог создать объект устройства для представления устройства. Инфраструктура продолжает вызывать функции обратного вызова драйвера, продвигаясь вверх по последовательности до тех пор, пока устройство не будет приведено с состоянием полной рабочей готовности. Необходимо иметь в виду, что перечисленные в рис. 7.3 функции обратного вызова для событий приводятся в восходящем порядке, т. е. функция *EvtDeviceFilterRemoveResourceRequirements* вызывается перед функцией *EvtDeviceFilterAddResourceRequirements* и т. д.

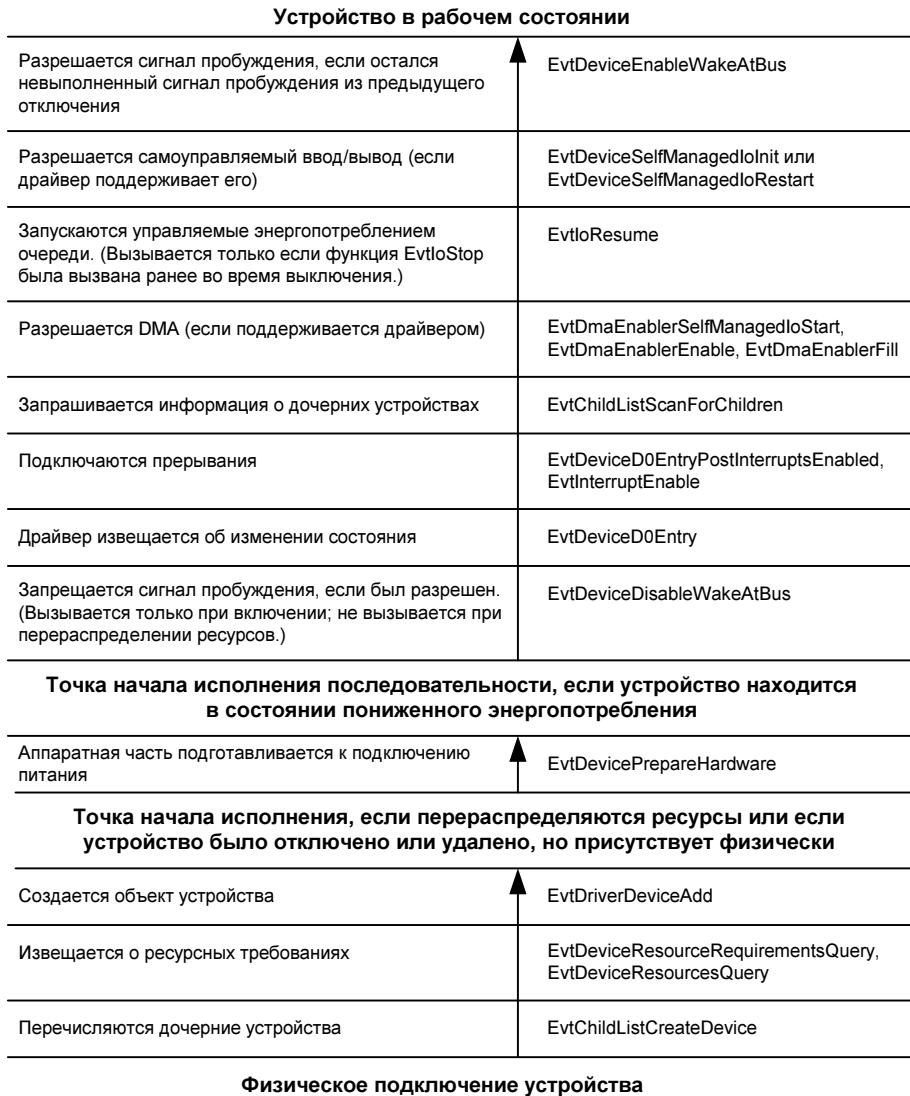
На рис. 7.4 показана последовательность обратных вызовов функций для драйвера шины (объект PDO), которые принимают участие в приведении устройства в полностью рабочее состояние.

Инфраструктура сохраняет объект PDO до тех пор, пока соответствующее физическое устройство не удалено из системы физически или не отключена шина, которая перечислила это

устройство. Например, если пользователь отключает устройство в Диспетчере устройств, но не удаляет его физически, KMDF сохраняет объект PDO. Это требование накладывается на нижележащей моделью WDM. Таким образом, три действия внизу рис. 7.4 выполняются только при перечислении Plug and Play, т. е. при первоначальной загрузке системы, когда пользователь физически подключает новое устройство и когда шина, к которой устройство подключено, перечисляет свои дочерние устройства.

<b>Устройство в рабочем состоянии</b>	
Разрешается самоуправляемый ввод/вывод (если драйвер поддерживает его)	EvtDeviceSelfManagedIoInit или EvtDeviceSelfManagedIoRestart
Запускаются управляемые энергопотреблением очереди. (Вызывается только если функция EvtIoStop была вызвана ранее во время выключения)	EvtIoResume
Запрещается сигнал пробуждения, если был разрешен. (Вызывается только при включении; не вызывается при перераспределении ресурсов)	EvtDeviceDisarmWakeFromSx EvtDeviceDisarmWakeFromS0
Запрашивается информация о дочерних устройствах	EvtChildListScanForChildren
Разрешается DMA (если поддерживается драйвером)	EvtDmaEnablerSelfManagedIoStart EvtDmaEnablerEnable EvtDmaEnablerFill
Подключаются прерывания	EvtDeviceD0EntryPostInterruptsEnabled EvtInterruptEnable
Драйвер извещается об изменении состояния	EvtDeviceD0Entry
<b>Точка начала исполнения последовательности, если устройство находится в состоянии пониженного энергопотребления</b>	
Аппаратная часть подготавливается к подключению питания	EvtDevicePrepareHardware
Изменения списка требуемых ресурсов	EvtDeviceRemoveAddedResources EvtDeviceFilterAddResourceRequirements EvtDeviceFilterRemoveResourceRequirements
<b>Точка начала исполнения при перераспределении ресурсов</b>	
Создается объект устройства	EvtDriverDeviceAdd
<b>Физическое подключение устройства</b>	

**Рис. 7.3.** Последовательность шагов и соответствующие функции обратного вызова при перечислении и запуске объекта FDO или FiDO KMDF



**Рис. 7.4.** Последовательность шагов при добавлении/запуске для объекта PDO

## Отключение питания и удаление устройства

При отключении питания, как и при включении устройства, последовательность обратных вызовов зависит от роли, играемой объектом устройства. В общем случае, последовательность действий при отключении питания и удаления устройства состоит из вызовов соответствующих "отменяющих" функций обратного вызова в порядке, обратном тому, в котором инфраструктура вызывала функции для приведения устройства в рабочее состояние. Драйверы выполняют операции, связанные с отключением, сверху вниз, поэтому драйвер на

верху стека устройства выполняет свои задачи первым, а объект PDO выполняет свои последним.

На рис. 7.5 показана последовательность действий и соответствующих обратных вызовов UMDF при отключении и удалении устройства. Исполнение последовательности начинается сверху, с устройством, находящимся в рабочем состоянии энергопотребления (D0).



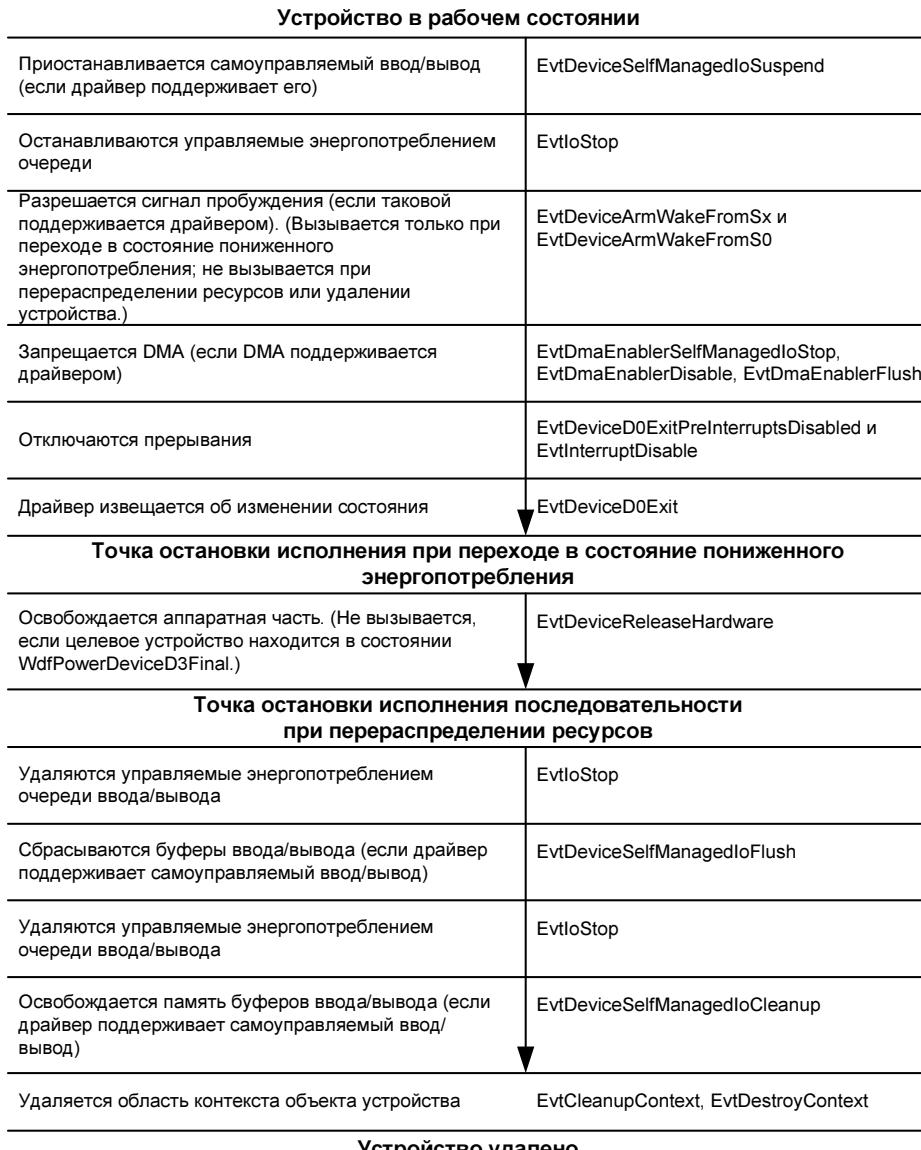
**Рис. 7.5.** Последовательность шагов при отключении и удалении устройства для драйвера UMDF

На рис. 7.6 показана последовательность действий и соответствующих обратных вызовов при отключении и удалении устройства для объекта FDO или FiDO KMDF.

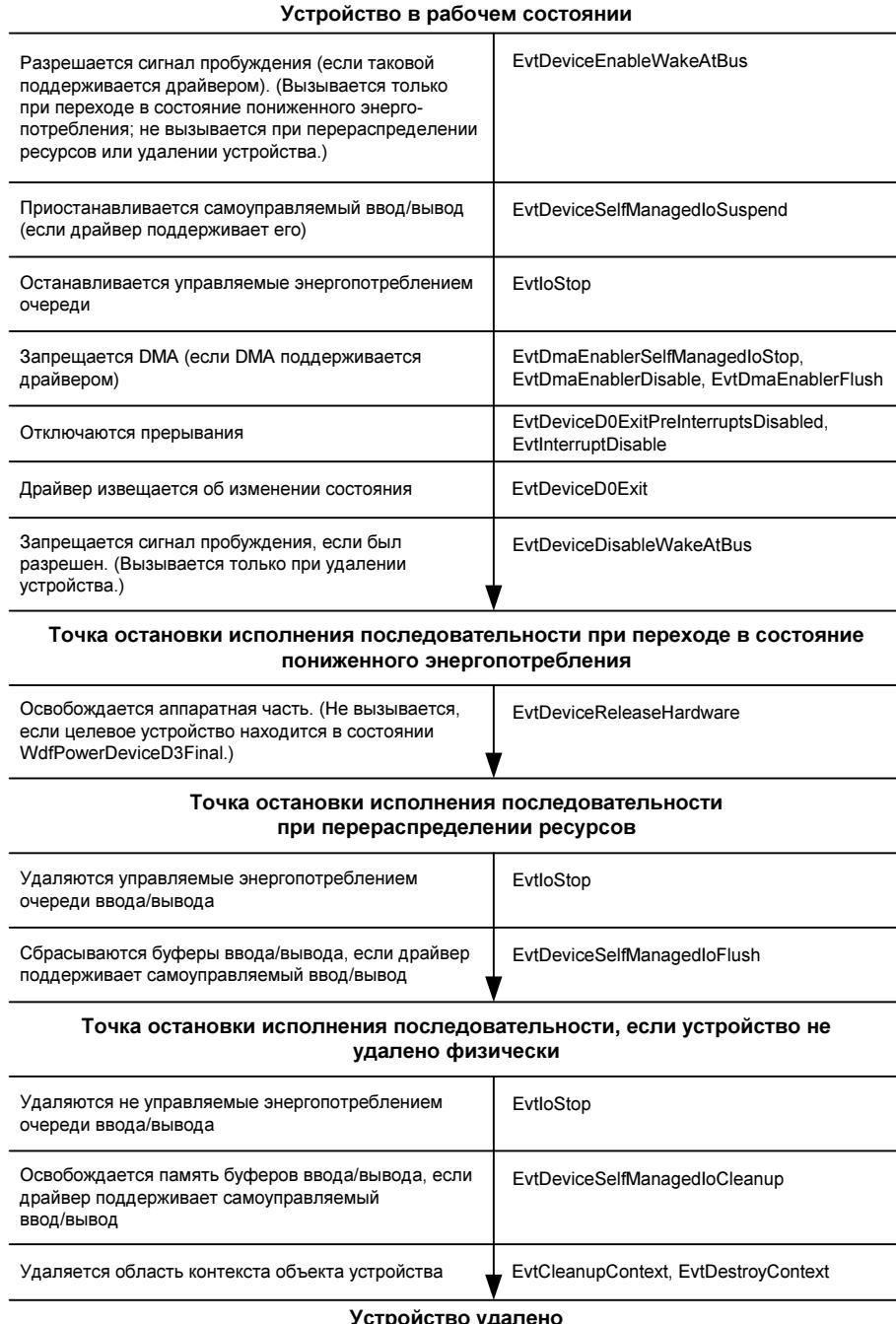
На рис. 7.7 показана последовательность действий и соответствующих обратных вызовов при отключении и удалении устройства для объекта PDO.

Как было упомянуто в разд. "Перечисление и запуск устройств" ранее в этой главе, инфраструктура сохраняет объект PDO до тех пор, пока соответствующее физическое устройство не удалено из системы физически или не отключена шина, которая перечислила это устрой-

ство. Например, если пользователь отключает устройство в Диспетчере устройств или исполняет утилиту для безопасного удаления устройства, но не удаляет его физически, KMDF сохраняет объект PDO. Если впоследствии устройство снова включается, KMDF использует тот же самый объект PDO и начинает выполнение последовательности включения устройства с вызова функции *EvtDevicePrepareHardware*, как было показано на рис. 7.4.



**Рис. 7.6.** Последовательность шагов при отключении и удалении устройства для объектов FDO и FiDO KMDF



**Рис. 7.7.** Последовательность шагов при отключении питания и удалении устройства для объекта PDO

## Неожиданное извлечение

Когда пользователь удаляет устройство, просто извлекая его из разъема, не отключив его предварительно в Диспетчере устройств или с помощью утилиты Windows для безопасного удаления устройства, принято говорить о неожиданном удалении устройства. При неожиданном удалении устройства WDF выполняет последовательность действий, слегка отличающуюся от последовательности, выполняемой при плановом удалении. WDF также выполняет последовательность действий для внезапного удаления, если любой драйвер в стеке режима ядра устанавливает состояние устройства недействительным, даже если устройство продолжает физически присутствовать. Драйвер KMDF может установить устройство недействительным с помощью метода `WdfDeviceSetDeviceState`.

Драйверы для всех удаляемых устройств должны обеспечивать, что функции обратного вызова для последовательностей, как подключения, так и отключения устройства, могут спрашиваться со сбоями, особенно со сбоями, вызываемыми удалением аппаратной части.

### Последовательность действий при неожиданном удалении для драйверов UMDF

О неожиданном удалении устройства UMDF извещает драйвер вызовом метода `IPnpCallback::OnSurpriseRemoval`. Этот метод может вызываться в любом порядке по отношению других методов обратного вызова в последовательности действий по удалению устройства.

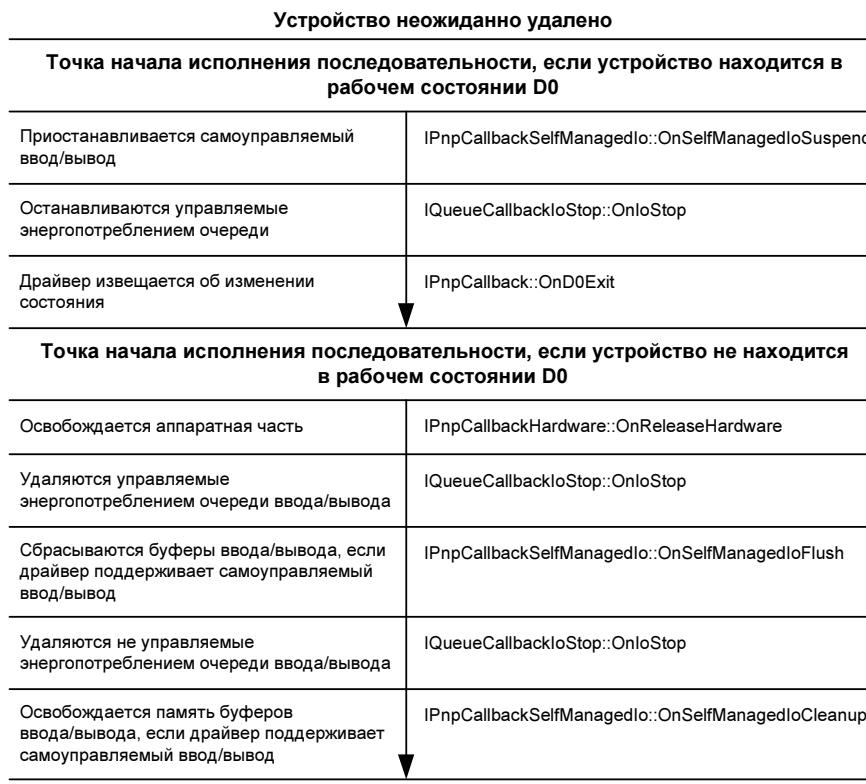
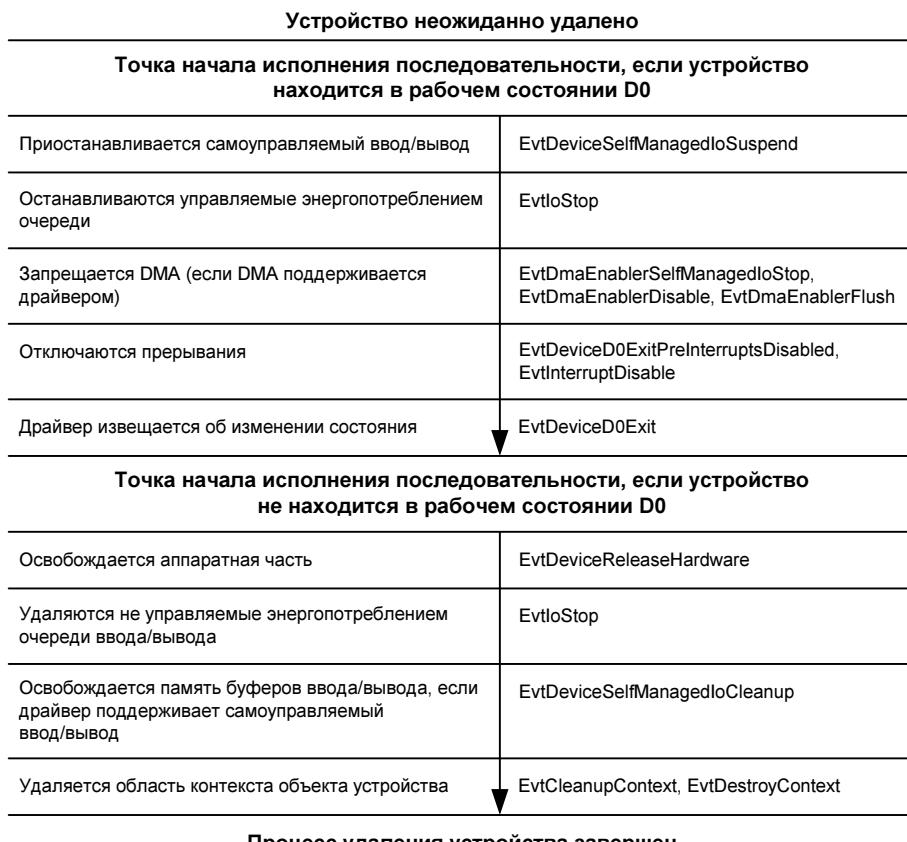


Рис. 7.8. Последовательность действий при неожиданном удалении устройства для драйвера UMDF

Обычно драйвер должен избегать обращений к аппаратному обеспечению, находящемуся в пути удаления. Но если при попытке драйвера получить доступ к аппаратной части он зависнет, по истечению определенного тайм-аута времени рефлектор снимет этот драйвер с выполнения. Последовательность действий при неожиданном удалении устройства для драйвера UMDF показана на рис. 7.8.

### **Последовательность действий при неожиданном удалении устройства для драйверов KMDF**

Инфраструктура может вызвать функцию обратного вызова *EvtDeviceSurpriseRemoval* в любое время до начала исполнения последовательности действий для выключения питания устройства, во время исполнения этой последовательности, и даже после его завершения. Например, если пользователь физически отключит устройство, когда оно находится в процессе перехода в состояние пониженного энергопотребления, инфраструктура может вызвать функцию *EvtDeviceSurpriseRemoval* посередине процедуры перехода. Не существует единого установленного порядка, в котором функция *EvtDeviceSurpriseRemoval* вызывается по отношению к другим функциям обратного вызова, применяемым для выключения устройства.



**Рис. 7.9.** Последовательность действий при неожиданном удалении устройства для драйверов KMDF

KMDF уничтожает объект устройства после возвращения управления функцией обратного вызова *EvtDeviceSurpriseRemoval* и закрытия всех дескрипторов объекта устройства WDF.

Необходимо избегать зависаний, вызванных неудачными попытками обращения к аппарату-ре. Для этого накладывается временное ограничение на длительность попытки доступа или реализуется таймер WDT.

Последовательность действий при неожиданном удалении устройства для драйверов KMDF показана на рис. 7.9.

## Реализация Plug and Play и управления энергопотреблением в драйверах WDF

В оставшейся части этой главы предоставляются образцы кода, в которых показывается реализация Plug and Play и управления энергопотреблением для разных типов драйверов. А именно, для:

- ◆ драйверов для чисто программных устройств;
- ◆ простые функциональные драйверы, такие как, например, протокольные функциональные драйверы UMDF и аппаратные функциональные драйверы KMDF, не поддерживающие режим ожидания или пробуждение;
- ◆ аппаратные функциональные драйверы KMDF, поддерживающие режим ожидания и пробуждение.

Сложность реализации Plug and Play и управления энергопотреблением возрастает для каждого следующего типа драйвера, и каждый следующий тип основывается на информации из предыдущих примеров. Таким же пошаговым способом вы можете добавлять код для обработки событий Plug and Play и энергопотребления в ваши собственные драйверы. Даже если ваше устройство поддерживает такие продвинутые возможности, как режим ожидания или пробуждение, разработку драйвера для него вы можете начать с реализации поддержки более простых возможностей устройства. Когда эти возможности работают должным образом, вы можете реализовать дополнительные функции обратного вызова для поддержки более сложных возможностей.

Описание каждого типа драйверов содержит следующее:

- ◆ тип драйвера и реализуемые им возможности Plug and Play и управления энергопотреблением;
- ◆ методы инфраструктуры, вызываемые драйвером, и функции обратного вызова для событий, реализуемые драйвером для поддержки возможностей Plug and Play и управления энергопотреблением;
- ◆ образец исходного кода, реализующий возможности драйвера;
- ◆ действия, предпринимаемые инфраструктурой в ответ на различные события Plug and Play и энергопотребления для данного типа драйвера.

Во всех листингах особо важные фрагменты кода выделены жирным шрифтом.

## Чисто программные драйверы Plug and Play и управление энергопотреблением

Чисто программный драйвер не управляет никаким аппаратным оборудованием, ни прямым, ни косвенным (например, посредством протокола USB) образом. Например, функциональный драйвер для устройства, являющегося дочерним для устройства системного корня (в дальнейшем — драйвер RED (Root-Enumerated Driver)) представляет собой чисто программный драйвер; некоторые драйверы фильтров также являются чисто программными драйверами. Узел devnode для функционального драйвера RED перечисляется относительно корня дерева устройств, и драйвер не сопоставлен ни с каким аппаратным оборудованием. Чисто программные драйверы можно создавать как для пользовательского режима, так и для режима ядра.

Чисто программный драйвер RED KMDF создает объект FDO и поэтому по умолчанию считается владельцем политики энергопотребления в своем стеке. Но т. к. этот драйвер не управляет никаким физическим оборудованием, он не выполняет никаких конкретных действий по политике энергопотребления — для управления политикой энергопотреблением достаточно стандартных установок WDF.

Драйверы фильтров редко владеют политикой энергопотребления в своих стеках. Но если драйвер фильтра является менеджером политики энергопотребления в своем стеке, он извещает инфраструктуру об этом в процессе инициализации объекта, чтобы WDF могла инициализировать объект устройства должным образом. Если же во всем остальном стандартные установки инфраструктуры являются приемлемыми для драйвера, ему не требуются никакие дополнительные функции обратного вызова для обработки событий Plug and Play и энергопотребления. Инфраструктура может управлять событиями Plug and Play и энергопотребления для драйвера фильтра точно так же, как она делает это для чисто программного функционального драйвера. Если же стандартных установок инфраструктуры недостаточно, драйвер фильтра может реализовать функции обратного вызова для обработки требуемых событий Plug and Play и энергопотребления.

По умолчанию инфраструктура реализует управление энергопотреблением для всех объектов очереди ввода/вывода, не являющихся дочерними объектами объектов FDO и PDO. Очереди, ассоциированные с объектами FiDO, не управляются энергопотреблением. Так как аппаратная часть устройства недоступна, когда устройство находится в любом состоянии энергопотребления, кроме состояния D0, инфраструктура отправляет запросы от энергоподляемых очередей драйверу только тогда, когда устройство находится в состоянии D0.

По определению, чисто программные драйверы не обращаются к аппаратному обеспечению устройства. Поэтому такие драйверы должны отключить возможность управляемости энергопотреблением для всех своих очередей, т. е. сделать их не управляемыми энергопотреблением. Отключение возможности управляемости энергопотреблением для очередей означает, что инфраструктура будет отправлять запросы драйверу независимо от состояния энергопотребления аппаратного обеспечения устройства. Тогда драйвер может обработать запрос обычным образом и, если необходимо, передать его следующему нижележащему драйверу. Драйвер отключает управляемость энергопотреблением для объекта очереди, когда он создает очередь.

Дополнительная информация об очередях предоставляется в главе 8.

### Совет

В главе 24 рассматривается вопрос, каким образом вставлять примечания в функции обратного вызова драйвера, чтобы SDV мог анализировать их на удовлетворение правил KMDF, требующим, чтобы драйвер, не являющийся владельцем политики энергопотребления, не мог вызывать следующие функции управления энергопотреблением: `WdfDeviceInitSetPowerPolicyEventCallbacks`, `WdfDeviceAssignSOIdleSettings` и `WdfDeviceAssignSxWakeSettings`.

## Пример UMDF: чисто программный драйвер фильтра для Plug and Play

Образец драйвера USB Filter не требует специального кода для обработки событий Plug and Play или управление энергопотреблением. Драйвер просто:

- ◆ инициализирует объект устройства как фильтр;
- ◆ указывает, что объект устройства не является владельцем политики энергопотребления. Этот шаг не является обязательным, т. к. UMDF по умолчанию полагает, что драйвер не является владельцем политики энергопотребления;
- ◆ создает не управляемые энергопотреблением очереди ввода/вывода.

Все эти действия выполняются в процессе исполнения метода `IDriverEntry::OnDeviceAdd`. В образце драйвера USB Filter исполняется метод `Initialize`, который реализован на объекте обратно вызова устройства в файле `Device.cpp`. Исходный код для этого метода приведен в листинге 7.1.

### Листинг 7.1. Пример инициализации Plug and Play в драйвере фильтра UMDF

```
HRESULT CMYDevice::Initialize(
    _in IWDFDriver           * FxDriver,
    _in IWDFDeviceInitialize * FxDeviceInit)
{
    IWDFDevice *fxDevice;
    HRESULT hr;
    FxDeviceInit->SetLockingConstraint(None);
FxDeviceInit->SetFilter();
    FxDeviceInit->SetPowerPolicyOwnership(FALSE);
    {
        IUnknown *unknown = this->QueryIUnknown();
        hr = FxDriver->CreateDevice(FxDeviceInit, unknown, &fxDevice);
        unknown->Release();
    }
    if (SUCCEEDED(hr)) {
        m_FxDevice = fxDevice;
        fxDevice->Release();
    }
    return hr;
}
```

Метод `Initialize` в листинге 7.1 инициализирует и создает инфраструктурный объект устройства. Код, выполняющий важные действия, выделен в листинге жирным шрифтом. Вызов метода `IWDFDeviceInitialize::SetFilter` информирует инфраструктуру о том, что драйвер

вер играет роль фильтра; поэтому инфраструктура должна модифицировать свои стандартные настройки для типов запросов, которые этот драйвер не обрабатывает. Вместо того чтобы прекращать выполнение таких запросов, инфраструктура передает их следующему нижележащему драйверу. Вызов метода `IWDFInitialize::SetPowerPolicyOwnership` информирует инфраструктуру о том, что драйвер не является владельцем политики энергопотребления устройства. Для этого драйвера исполнение данного метода не является обязательным; он был использован лишь в целях демонстрации.

Другим единственным необходимым шагом в драйвере фильтра UMDF является создание не управляемых энергопотреблением очередей. Драйвер выполняет эту задачу, вызывая метод `IWDFDevice::CreateIoQueue`, как показано далее:

```
hr = FxDevice->CreateIoQueue( unknown,
                               TRUE, // bDefaultQueue
                               WdfIoQueueDispatchParallel,
                               FALSE, // bPowerManaged
                               TRUE, // bAllowZeroLengthRequests
                               &fxQueue
                           );
```

В этом вызове важным является четвертый параметр (`bPowerManaged`), посредством которого указывается, должна ли инфраструктура управлять энергопотреблением для очередей. Чисто программный драйвер передает в этом параметре значение `FALSE`, чтобы инфраструктура отправляла запросы драйверу, независимо от того, находится или нет устройство в рабочем состоянии энергопотребления.

## Пример KMDF: чисто программный драйвер фильтра для Plug and Play

Как было описано ранее, по умолчанию инфраструктура автоматически обрабатывает события Plug and Play и управления энергопотреблением для чисто программных драйверов.

В листинге 7.2, который был адаптирован с образца драйвера Toaster Filter, показана базовая функция `EvtDriverDeviceAdd` для чисто программного драйвера фильтра KMDF. Эта функция организует две возможности Plug and Play или управления энергопотреблением, которые выделяются жирным шрифтом в листинге:

- ◆ необязательную функцию очистки для объекта устройства;
- ◆ не управляемую энергиопотреблением очередь ввода/вывода.

### Листинг 7.2. Пример инициализации Plug and Play в чисто программном драйвере фильтра KMDF

```
NTSTATUS FilterEvtDriverDeviceAdd(
    IN WDFDRIVER Driver,
    IN PWDDEVICE_INIT DeviceInit)
{
    NTSTATUS status = STATUS_SUCCESS;
    PFDO_DATA fdoData;
    WDF_IO_QUEUE_CONFIG queueConfig;
    WDF_OBJECT_ATTRIBUTES fdoAttributes;
    WDFDEVICE hDevice;
```

```

WdfFdoInitSetFilter(DeviceInit);
// Инициализируются атрибуты объекта для WDFDEVICE.
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&fdoAttributes, FDO_DATA);
fdoAttributes.EvtCleanupCallback = FilterEvtDeviceContextCleanup;
// Создается инфраструктурный объект устройства.
status = WdfDeviceCreateC&DeviceInit, &fdoAttributes, &hDevice);
if (!NT_SUCCESS(status)) {
    return status;
}
status = WdfDeviceCreateDeviceInterface(hDevice,
                                         &GUID_DEVINTERFACE_FILTER,
                                         NULL);
if (!NT_SUCCESS(status)) {
    return status;
}
// Инициализируется очередь по умолчанию.
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&queueConfig,
                                       WdfIoQueueDispatchParallel);
queueConfig.PowerManaged = FALSE;
// Указываются функции обратного вызова для обработки событий.
queueConfig.EvtIoWrite = FilterEvtIoWrite;
// Создается очередь.
status = WdfIoQueueCreate(hDevice, &queueConfig,
                           WDF_NO_OBJECT_ATTRIBUTES, NULL);
if (!NT_SUCCESS(status)) {
    return status;
}
return STATUS_SUCCESS;
}

```

В листинге при вызове функции *EvtDriverDeviceAdd* передаются в качестве параметров указатель на объект драйвера и указатель на структуру *WDFDEVICE\_INIT*. Структура *WDFDEVICE\_INIT* используется для инициализации различных характеристик, которые применяются при создании объекта устройства.

Драйвер информирует инфраструктуру о том, что объект устройства представляет фильтр, передавая указатель *WDFDEVICE\_INIT* методу *WdfFdoInitSetFilter*. В результате, инфраструктура модифицирует стандартную обработку для всех очередей ввода/вывода, являющимися дочерними объектами объекта устройства. Вместо того чтобы не выполнять не обрабатываемые драйвером запросы, инфраструктура передает их следующему нижележащему драйверу. Кроме этого, инфраструктура создает очереди ввода/вывода, не управляемые энергопотреблением.

Драйвер регистрирует тип контекста объекта устройства (*FDO\_DATA*), как часть структуры *WDF\_OBJECT\_ATTRIBUTES*. Заполняя член *EvtCleanupCallback* этой же структуры, драйвер регистрируется для вызова своей функции *FilterEvtDeviceContextCleanup* по удалению объекта устройства. Драйвер должен реализовывать этот обратный вызов, если, например, он выделил иную память, чем предоставляемую стандартными структурами контекста объекта WDF, которую нужно освободить при удалении объекта.

Потом драйвер создает объект устройства и интерфейс устройства, вызывая для этого методы *WdfDeviceCreate* и *WdfDeviceCreateDeviceInterface* соответственно.

После этого, драйвер инициализирует структуру *WDF\_IO\_QUEUE\_CONFIG* для своей стандартной очереди, предоставляя обратный вызов для обработки запросов на чтение. Он устанавливает

значение поля `PowerManaged` структуры `WDF_IO_QUEUE_CONFIG` в `FALSE`, чтобы указать, что создаваемая очередь ввода/вывода должна быть не управляемой энергопотреблением. Драйвер передает эту структуру в качестве ввода методу `WdfIoQueueCreate`, для создания одной стандартной очереди для обработки запросов для драйвера.

Создание не управляемой энергопотреблением очереди означает, что в любое время по прибытию запроса на запись инфраструктура вызывает драйвер, независимо от состояния энергопотребления устройства.

## Действия инфраструктуры для чисто программных драйверов

В чисто программных драйверах и драйверах фильтров инфраструктура обрабатывает почти все операции, связанные с Plug and Play и управлением энергопотреблением. Так как драйвер не управляет никаким аппаратным обеспечением, от него не требуется предоставлять каких-либо дополнительных функций обратного вызова. Инфраструктура автоматически обрабатывает должным образом все запросы управления энергопотреблением.

Единственной функцией обратного вызова для событий Plug and Play в драйвере UMDF является `IDriverEntry::OnDeviceAdd`, а в драйвере KMDF — `EvtDriverDeviceAdd`. Хотя в примере драйвера KMDF также реализуется необязательная функция `EvtCleanupCallback`, какой-либо необходимости для предоставления этой функции обратного вызова в предшествующем примере не существует.

Драйверы в примерах отключают управление энергопотреблением для всех своих очередей. В результате, инфраструктура прекращает автоматически останавливать и запускать очередь на основе прибывающих событий Plug and Play и управления энергопотреблением. Вместо этого, драйвер продолжает получать запросы ввода/вывода, независимо от состояния Plug and Play и энергопотребления устройства.

## Plug and Play и управление энергопотреблением в простых аппаратных драйверах

Драйвер, поддерживающий операции с аппаратным обеспечением (например, протокольный функциональный драйвер UMDF или аппаратный функциональный драйвер KMDF), отличается от чисто программного драйвера в следующем.

- ◆ Драйвер, поддерживающий аппаратное обеспечение, должен инициализировать свое устройство в известное состояние всякий раз, когда устройство переходит в состояние D0, включительно при запуске системы.

Обычно этим известным состоянием является состояние полного сброса. Если устройство поддерживает прерывания или DMA, прерывания отключаются, а DMA останавливаются.

- ◆ Большинство драйверов, взаимодействующих непосредственно с аппаратным обеспечением своего устройства, создают одну или несколько управляемых энергопотреблением очередей ввода/вывода.

Инфраструктура автоматически прекращает отправлять запросы из управляемых энергопотреблением очередей, когда устройство недоступно, например, находится в состоянии пониженного энергопотребления.

Простой аппаратный драйвер, как описано в этом разделе, управляет аппаратным обеспечением своего устройства, выполняя инициализацию при включении, деинициализацию при выключении и используя управляемые энергопотреблением очереди.

Большинство аппаратных драйверов KMDF управляют аппаратными ресурсами и прерываниями своих устройств, поэтому они должны поддерживать функции обратного вызова для обработки ресурсов и разрешения, запрещения и обработки прерываний.

Информация о том, как реализовать DMA в драйвере KMDF, приводится в главе 17. А в главе 16 описывается код для обработки прерываний в драйверах KMDF.

Продвинутые возможности, такие как режим ожидания и пробуждение, поддерживаются только в KMDF и описываются в разд. "Продвинутые возможности управления энергопотреблением для драйверов KMDF" далее в этой главе.

## Инициализация и деинициализация устройства при включении и выключении

Инфраструктура предоставляет функциональным драйверам аппаратного обеспечения возможность выполнять инициализацию всегда, когда устройство переходит в состояние D0 и деинициализацию, когда устройство покидает состояние D0. Всякий раз, когда устройство переходит в состояние энергопотребления D0 (рабочий режим), инфраструктура вызывает соответствующую функцию обратного вызова:

- ◆ для драйверов UMDF инфраструктура вызывает функцию `IPnpCallback::OnD0Entry`;
- ◆ для драйверов KMDF инфраструктура вызывает функцию `EvtDeviceD0Entry`.

Функция `EvtDeviceD0Entry` вызывается перед вызовом функции `EvtInterruptEnable`. Поэтому в этот момент прерывания для устройства еще не были разрешены и устройство еще не подключено к обратному вызову `EvtInterruptIsr` драйвера. Во время исполнения функции `EvtDeviceD0Entry` драйверы не должны разрешать прерывания на своих устройствах или выполнять какие-либо действия, которые могли бы вызвать прерывания на их устройствах. Это важное условие для предотвращения возможного "шторма прерываний". То же самое относится и к обратному вызову функции `EvtDevicePrepareHardware`. Если устройство необходимо инициализировать после того, как его прерывание было подключено, драйвер должен зарегистрировать функцию обратного вызова `EvtDeviceD0EntryPostInterruptsEnabled`.

Инфраструктура вызывает эти функции после того, как драйвер шины привел устройство в рабочее состояние D0, таким образом, делая аппаратное обеспечение устройства доступным для драйвера. В порядке презумпции все устройства приводятся в рабочее состояние D0 при первоначальном обнаружении устройства, например при запуске системы, и при перезапуске устройства после его остановки менеджером PnP для перераспределения ресурсов. Поэтому, инфраструктура всегда вызывает функции обратного вызова для события перехода в состояние D0 при запуске, после вызова функций для подготовки аппаратного обеспечения.

В процедуре обратного вызова для события входа в состояние D0 драйвер выполняет любые требуемые задачи по обслуживанию аппаратного обеспечения при каждом входе устройства в состояние D0. Такими задачами могут быть загрузка микропрограммного обеспечения в устройства и инициализация устройства в известное состояние или восстановление состояния, сохраненное ранее при отключении питания.

Непосредственно перед каждым выходом устройства из состояния D0 инфраструктура вызывает функцию обратного вызова драйвера для события выхода из состояния D0:

- ◆ для драйверов UMDF инфраструктура вызывает функцию `IPnpCallback::OnD0Exit`;
- ◆ для драйверов KMDF инфраструктура вызывает функцию `EvtDeviceD0Exit`.

Для драйверов KMDF инфраструктура вызывает функцию `EvtDeviceD0Exit` после вызова функции `EvtInterruptDisable`, так что прерывания были запрещены и отключены от функции `EvtInterruptIsr` драйвера. Подобно ситуации с переходом в состояние D0, если требуется очистка устройства перед тем, как запретить его прерывания, драйвер должен зарегистрировать функцию обратного вызова `EvtDeviceD0ExitPreInterruptsDisabled`.

При обработке выхода из состояния D0 драйвер выполняет задачи, связанные с понижением энергопотребления, такие как, например, сохранение внутреннего состояния устройства. При исполнении этих обратных вызовов аппаратное обеспечение устройства продолжает быть доступным, т. к. устройство все еще находится в состоянии D0.

## Управление энергопотреблением очередей в аппаратных функциональных драйверах

По умолчанию инфраструктура управляет энергопотреблением для всех объектов очереди ввода/вывода, являющихся дочерними объектами объектов FDO и PDO. Как было описано ранее, когда инфраструктура управляет энергопотреблением очереди, она отправляет запросы из очереди функциям обратного вызова ввода/вывода драйвера только тогда, когда аппаратное обеспечение устройства доступно и находится в состоянии D0. Предоставление управления энергопотреблением очереди инфраструктуре означает, что драйверу не требуется содержать информацию о состоянии устройства или проверять состояние устройства каждый раз, когда он получает запрос ввода/вывода из очереди. Вместо этого он может обработать запрос и обращаться к аппаратному обеспечению устройства как требуется для полной обработки запроса.

Конечно же, не все запросы ввода/вывода, получаемые драйвером, требуют доступа к аппаратному обеспечению устройства. Например, драйвер может обрабатывать некоторые запросы IOCTL, не обращаясь к аппаратному обеспечению. Такой драйвер должен создать две очереди — одну управляемую энергопотреблением, и одну — неуправляемую. Драйвер конфигурирует не управляемую энергопотреблением очередь для получения всех запросов IOCTL для устройства от инфраструктуры. Инфраструктура отправляет запросы из этой очереди независимо от состояния энергопотребления устройства. Драйвер анализирует каждый запрос, обрабатывает запрос, если может, и, если устройство не находится в рабочем состоянии, пересыпает все запросы, которые он не может обработать, управляемой энергопотреблением очереди.

Драйверы обычно создают и конфигурируют свои очереди ввода/вывода во время процесса добавления устройства, т. е. в обратном вызове метода `IDriverEntry::OnDeviceAdd` для драйверов UMDF или в обратном вызове функции `EvtDriverDeviceAdd` для драйверов KMDF.

Драйверы отключают управляемость энергопотреблением для очереди следующим образом:

- ◆ при создании очереди драйверы UMDF устанавливают значение параметра `bPowerManaged` метода `IWDFDevice::CreateIoQueue` в FALSE;
- ◆ при создании очереди драйверы KMDF устанавливают значение поля `PowerManaged` структуры `WDF_IO_QUEUE_CONFIG` в WdfFalse.

Драйверы, обрабатывающие запросы ввода/вывода, одни из которых требуют доступа к аппаратной части, а другие нет, должны создавать несколько очередей и сортировать запросы на этой основе. Образец UMDF-драйвера Fx2\_Driver и образец KMDF-драйвера Osrusbf2 создают как управляемые, так и не управляемые энергопотреблением очереди.

## Пример UMDF: код для Plug and Play и управления энергопотреблением в протокольном функциональном драйвере

Функциональный драйвер UMDF, управляющий аппаратным обеспечением устройства, отличается от чисто программного драйвера в нескольких отношениях:

- ◆ драйвер обычно создает одну или несколько управляемых энергопотреблением очередей;
- ◆ драйвер реализует интерфейсы `IPnpCallback` и `IPnpCallbackHardware` в соответствии с требованиями объекта обратного вызова устройства для выполнения задач, связанных с Plug and Play и состоянием энергопотребления устройства;
- ◆ драйвер должен определить, должен ли он быть владельцем политики энергопотребления для стека устройства.

Драйвер UMDF не может быть владельцем политики энергопотребления в стеке устройства USB. Если стек устройства USB содержит драйвер UMDF, владельцем политики энергопотребления всегда будет `WinUSB.sys`.

Если драйвер UMDF находится в стеке устройства иного, чем USB, владельцем политики энергопотребления обычно является драйвер режима ядра, т. к. драйверы режимы ядра могут поддерживать режим ожидания и пробуждения, в то время как драйверы UMDF этого делать не могут. Но если устройству не требуется поддержка режима ожидания или пробуждения, следует рассмотреть возможность сделать владельцем политики энергопотребления драйвер UMDF.

В табл. 7.3 и 7.4 приводятся краткие описания методов интерфейсов `IPnpCallbackHardware` и `IPnpCallback`.

**Таблица 7.3. Методы интерфейса `IPnpCallbackHardware`**

Метод	Описание	Когда вызывается
<code>OnPrepareHardware</code>	Подготавливает устройство и драйвер к переходу в рабочее состояние после перечисления или после перераспределения ресурсов	После возвращения управления методом <code>IDriverEntry::OnDeviceAdd</code> и перед тем, как устройство переходит в рабочее состояние энергопотребления
<code>OnReleaseHardware</code>	Подготавливает устройство и драйвер к выключению системы или к перераспределению ресурсов	После выхода устройства из рабочего состояния энергопотребления, но перед удалением его очередей

Методы интерфейса `IPnpCallbackHardware` предоставляют возможность драйверу выполнять операции при добавлении или удалении его устройства из системы или при перераспределении ресурсов.

**Таблица 7.4.** Методы интерфейса *IPnpCallback*

Метод	Описание	Когда вызывается
OnD0Entry	Исполняет операции для подготовки устройства к началу работы	Сразу же после того, как устройство переходит в рабочее состояние энергопотребления
OnD0Exit	Исполняет операции для подготовки устройства к окончанию работы	Непосредственно перед тем, как устройство покидает рабочее состояние энергопотребления
OnSurpriseRemoval	Выполняет операции очистки после неожиданного удаления устройства	Сразу же после неожиданного удаления устройства
OnQueryRemove	Предоставляет драйверу возможность отказать в выполнении запроса на удаление устройства	При пребывании устройства в рабочем состоянии, перед физическим удалением устройства
OnQueryStop	Предоставляет драйверу возможность отказать в выполнении запроса на остановку устройства для перераспределения ресурсов	При пребывании устройства в рабочем состоянии, перед его остановкой для перераспределения ресурсов

Интерфейс *IPnpCallback* содержит методы, необходимые для поддержки наиболее распространенных событий Plug and Play и энергопотребления, например, выполнение любой инициализации, требуемой устройством после включения, и соответствующей деинициализации, требуемой перед понижением или полной остановкой энергопотребления устройства.

Образец драйвера *Fx2\_Driver* представляет собой протокольный функциональный драйвер, который реализует как интерфейс *IPnpCallback*, так и интерфейс *IPnpCallbackHardware* на объекте обратного вызова устройства. Драйвер создает стандартную управляемую энергопотреблением очередь для запросов на чтение и запись и отдельную управляемую энергопотреблением очередь только для запросов IOCTL.

### Управляемая энергопотреблением очередь для драйвера UMDF

Для создания управляемой энергопотреблением очереди драйвер UMDF передает значение TRUE для параметра *bPowerManaged* методу *IWDFDevice::CreateIoQueue*, как показано в следующем фрагменте кода:

```
hr = FxDevice->CreateIoQueue( unknown,
                               TRUE, // bDefaultQueue
                               WdfIoQueueDispatchParallel,
                               TRUE, // bPowerManaged
                               TRUE, // bAllowZeroLengthRequests
                               &fxQueue
                             );
```

Драйвер обычно создает очереди ввода/вывода в обратном вызове метода *IDriverEntry::OnDeviceAdd*.

### Методы интерфейса *IPnpCallbackHardware*

Инфраструктура вызывает методы интерфейса *IPnpCallbackHardware* объекта устройства до входа устройства в состояние D0 и после его выхода из состояния D0.

В методе `OnPrepareHardware` драйвер подготавливается к взаимодействию с аппаратным обеспечением устройства. Он открывает дескриптор устройства и вызывает внутренние функции, чтобы получить информацию об интерфейсах и конечных точках USB.

Метод `OnPrepareHardware`, который драйвер `Fx2_Driver` реализует на объекте обратного вызова, определен в файле `Device.cpp`. Исходный код определения показан в листинге 7.3. В целях экономии места код для обработки ошибок был опущен.

### Листинг 7.3. Метод `IPnpCallbackHardware::OnPrepareHardware` образца драйвера

```

HRESULT CMyDevice::OnPrepareHardware(
    _in IWDFDevice * /* FxDevice */
)
{
    HRESULT hr;
    . . . // Код опущен.
    // Создаются и конфигурируются исполнители ввода/вывода USB.
    hr = CreateUsbToTargets();
    if (SUCCEEDED(hr)) {
        ULONG length = sizeof(m_Speed);
        hr = m_pIUsbTargetDevice->RetrieveDeviceInformation(DEVICE_SPEED,
                                                               &length,
                                                               &m_Speed);
        if (FAILED(hr)) {
            // Генерируется сообщение трассировки.
        }
    }
    . . . // Код опущен.
    hr = ConfigureUsbPipes();
    // Инициализируются настройки управления
    // энергопотребления устройства.
    if (SUCCEEDED(hr)) {
        hr = SetPowerManagement();
    }
    if (SUCCEEDED(hr)) {
        . . . // Код опущен.
    }
    if (FAILED(hr)) {
        ClearTargets();
    }
    return hr;
}

```

Метод `OnPrepareHardware` исполняет операции, требуемые для подготовки устройства к выполнению ввода/вывода, до перехода устройства в рабочее состояние. Сюда входит создание и конфигурирование исполнителей ввода/вывода USB для драйвера. Информацию об аппаратном обеспечении USB метод `OnPrepareHardware` получает с помощью интерфейса инфраструктуры `IWDFUsbTargetDevice`.

Прежде чем устройство USB может перейти в состояние D0, драйвер устанавливает его политику энергопотребления. Для этого применяется вспомогательная функция `SetPowerManagement`, которая использует специфичные для USB методы интерфейса `IWDFUsbTargetDevice`.

В листинге 7.4 приведен метод OnReleaseHardware образца драйвера Fx2\_Driver.

#### Листинг 7.4. Метод IPnpCallbackHardware::OnPrepareHardware

```
HRESULT CMYDevice::OnReleaseHardware(
    _in IWDFDevice * /* FxDevice */)
{
    ClearTargets();
    return S_OK;
}
```

Метод OnPrepareHardware исполняет операции, требуемые при выходе устройства из рабочего состояния. Драйвер Fx2\_Driver освобождает все ссылки драйвера на объекты исполнителей ввода/вывода, что является единственной задачей, выполняемой вспомогательной функцией ClearTargets. Никакого дополнительного обслуживания аппаратного обеспечения перед началом снижения энергопотребления не требуется. Например, драйвер не сохраняет никакой информации о контексте аппаратного обеспечения.

### Методы интерфейса IPnpCallback

При включении питания обучающего устройства OSR USB Fx2 драйвер Fx2\_Driver запускает его каналы исполнителя ввода/вывода USB. При выключении питания устройства драйвер останавливает каналы исполнителей. Для выполнения этих задач драйвер реализует методы OnD0Entry и OnD0Exit интерфейса IPnpCallback. Кроме этого, в случае неожиданного удаления устройства пользователем драйвер удаляет исполнителей, для чего реализуется метод OnSurpriseRemoval. Остальные методы интерфейса IPnpCallback реализованы в этом драйвере лишь формально.

В образце драйвера эти три метода всего лишь выполняют операции с исполнителями ввода/вывода, которые драйвер запускает при переходе в рабочее состояние и останавливает при выходе из него. Драйвер на самом деле не манипулирует никаким аппаратным обеспечением ни в одной из этих функций. Он лишь осуществляет приготовления для начала и окончания обработки запросов ввода/вывода.

Дополнительная информация об исполнителях ввода/вывода устройств USB приводится в главе 9.

В листингах 7.5—7.7 приводится исходный код методов OnD0Entry, OnD0Exit и OnSurpriseRemoval из файла Device.cpp.

#### Листинг 7.5. Метод IPnpCallback::OnD0Entry

```
HRESULT STDMETHODCALLTYPE
CMYDevice::OnD0Entry(
    _in IWDFDevice * /* FxDevice */,
    _in WDF_POWER_DEVICE_STATE /* PreviousState */)
{
    StartTarget(m_pIUsbInterruptPipe);
    return InitiatePendingRead();
}
```

В методе `OnD0Entry` драйвер запускает одного из исполнителей ввода/вывода, которых он создал в методе `IPnpCallbackHardware::OnPrepareHardware`, и инициирует для этого исполнителя запрос на чтение. Для выполнения этих двух задач применяются вспомогательные функции `StartTarget` и `InitiatePendingRead`.

#### Листинг 7.6. Метод `IPnpCallback::OnD0Exit`

```
HRESULT STDMETHODCALLTYPE
CMyDevice::OnD0Exit(
    _in IWDFDevice /* FxDevice */,
    _in WDF_POWER_DEVICE_STATE /* New/State */
)
{
    if (WdfIoTargetStarted == GetTargetState(m_pIUsbInterruptPipe) )
    {
        StopTarget(m_pIUsbInterruptPipe);
    }
    return S_OK;
}
```

Как можно видеть в листинге 7.6, метод `OnD0Exit` останавливает исполнителя ввода/вывода. Драйвер вызывает вспомогательную функцию `GetTargetState`, чтобы определить состояние исполнителя. Это вызвано тем, что если устройство было неожиданно удалено, тогда уже был исполнен метод `OnSurpriseRemoval` (см. листинг 7.7), который удалил исполнителя.

#### Листинг 7.7. Метод `IPnpCallback::OnSurpriseRemoval`

```
VOID STDMETHODCALLTYPE
CMyDevice::OnSurpriseRemoval(
    _in IWDFDevice /* FxDevice */
)
{
    RemoveTarget(m_pIUsbInterruptPipe, TRUE); //bSurpriseRemove
    return;
}
```

Если пользователь неожиданно физически отключит устройство, инфраструктура вызывает метод `OnSurpriseRemoval`, перед тем как активировать любой другой обратный вызов в последовательности выключения, как описано в разд. "Последовательность действий при неожиданном удалении для драйверов UMDF" ранее в этой главе. В образце драйвера `Fx2_Driver` эта функция удаляет исполнителя ввода/вывода.

## Пример KMDF: код для Plug and Play и управления энергопотреблением в простом аппаратном функциональном драйвере

Простой аппаратный функциональный драйвер KMDF, управляющий устройством в процессе включения и выключения, требует только несколько дополнительных функций обратного вызова, чем для чисто программного драйвера.

Код в этом разделе был адаптирован из образца драйвера Osrusbf2. Кроме возможностей, требуемых для чисто программного драйвера, в нем предоставляется поддержка для следующих возможностей:

- ◆ функции обратного вызова *EvtDevicePrepareHardware*;
- ◆ функциям обратного вызова *EvtDeviceD0Entry* и *EvtDeviceD0Exit*;
- ◆ четыре очереди ввода/вывода, три из которых управляемые энергопотреблением.

С помощью этих нескольких функций драйвер предоставляет полную поддержку Plug and Play и управления энергопотреблением для своего устройства.

## Пример KMDF: регистрация обратных вызовов и организация управляемых энергопотреблением очередей

В листинге 7.8 показывается регистрация драйвером Osrusbf2 базовых обратных вызовов для событий Plug and Play и управление энергопотреблением и создание управляемых энергопотреблением очередей ввода/вывода. Эта функция определена в исходном файле Device.c.

**Листинг 7.8. Функция EvtDriverDeviceAdd для простого аппаратного функционального драйвера**

```
NTSTATUS OsrfxEvtDeviceAdd(
    IN WDFDRIVER           Driver,
    IN PWDFDEVICE_INIT     DeviceInit)
{
    WDF_PNPPOWER_EVENT_CALLBACKS      pnpPowerCallbacks;
    WDF_OBJECT_ATTRIBUTES            attributes;
    NTSTATUS                         status;
    WDFDEVICE                        device;
    WDF_DEVICE_PNP_CAPABILITIES     pnpCaps;
    WDF_IO_QUEUE_CONFIG             ioQueueConfig;
    PDEVICE_CONTEXT                  pDevContext;
    WDFQUEUE                         queue;
    UNREFERENCED_PARAMETER(Driver);

    WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
    pnpPowerCallbacks.EvtDevicePrepareHardware =
        OsrfxEvtDevicePrepareHardware;
    pnpPowerCallbacks.EvtDeviceD0Entry = OsrfxEvtDeviceD0Entry;
    pnpPowerCallbacks.EvtDeviceD0Exit = OsrfxEvtDeviceD0Exit;
    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);
    WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoBuffered);
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, DEVICE_CONTEXT);
    // Создается инфраструктурный объект устройства.
    status = WdfDeviceCreateC&DeviceInit, &attributes, &device);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    pDevContext = GetDeviceContext(device);
    // Устанавливается SurpriseRemovalOK в структуре Device Capabilities,
    // чтобы предотвратить вывод всплывающего сообщения в Windows 2000
    // при неожиданном удалении устройства пользователем.
```

```

WDF_DEVICE_PNP_CAPABILITIES_INIT(&pnpCaps);
pnpCaps.SurpriseRemovalOK = WdfTrue;
WdfDeviceSetPnpCapabilities(device, &pnpCaps);
// Создается стандартная очередь.
. . . // Код опущен.

// Создается отдельная простая очередь для запросов на чтение.
WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig,
                         WdfIoQueueDispatchSequential);
ioQueueConfig.EvtIoRead = OsrFxEvtIoRead;
ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;
status = WdfIoQueueCreate(device,
                          &ioQueueConfig,
                          WDF_NO_OBJ ECT_ATTRIBUTES,
                          &queue // Дескриптор очереди.
                     );
if (!NT_SUCCESS(status)) {
    return status;
}
status = WdfDeviceConfigureRequestDispatching(device, queue,
                                               WdfRequestTypeRead);
if (!NT_SUCCESS(status)) {
    return status;
}
// Создается другая простая очередь для запросов на запись.
. . . // Код опущен.
// Создается управляемая вручную очередь ввода/вывода.
// Запросы из этой очереди извлекаются только тогда,
// когда устройство создает прерывание.
WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig,
                         WdfIoQueueDispatchManual);
ioQueueConfig.PowerManaged = WdfFalse;
status = WdfIoQueueCreate(device,
                          &ioQueueConfig,
                          WDF_NO_OBJ ECT_ATTRIBUTES,
                          &pDevContext->InterruptMsgQueue
                     );
if (!NT_SUCCESS(status)) {
    . . . // Дополнительный код опущен.
}
return status;
}

```

В предыдущем листинге первые выделенные жирным шрифтом строчки кода относятся к обратным вызовам для событий Plug and Play и управления энергопотреблением. Драйвер инициализирует структуру `WDF_PNPPOWER_EVENT_CALLBACKS` и заполняет ее указателями на свои функции обратного вызова `EvtDevicePrepareHardware`, `EvtDeviceD0Entry` и `EvtDeviceD0Exit`. Так как этот драйвер управляет устройством USB, он реализует обратный вызов только функции `EvtDevicePrepareHardware`, без соответствующего вызова функции `EvtDeviceReleaseHardware`. В драйверах USB функция обратного вызова `EvtDevicePrepareHardware` выбирает интерфейсы и получает другую информацию об устройстве USB, перед тем, как устройство переходит в рабочее состояние энергопотребления. Но соответствующая deinициализация не требуется, и поэтому драйвер не реализует обратный

вызов функции *EvtDeviceReleaseHardware*. Драйверы для иных, чем USB, устройств обычно реализуют обратные вызовы обеих этих функций.

После этого драйвер устанавливает структуру *WDF\_PNPPOWER\_EVENT\_CALLBACKS* в структуру *WDFDEVICE\_INIT*, вызывая для этого метод *WdfDeviceInitSetPnpPowerEventCallbacks*. Структура *WDFDEVICE\_INIT* и, соответственно, только что описанные обратные вызовы ассоциируются с объектом устройства, когда драйвер вызывает метод *WdfDeviceCreate*.

Второй фрагмент выделенного жирным шрифтом кода устанавливает возможности Plug and Play устройства. После создания объекта устройства драйвер инициализирует структуру *WDF\_DEVICE\_PNP\_CAPABILITIES*, устанавливая значение поля *SurpriseRemovalOK* в *WdfTrue*, после чего вызывает метод *WdfDeviceSetPnpCapabilities*, чтобы передать эту информацию инфраструктуре. Это значение указывает, что устройство можно безопасно извлекать, не применяя специальную утилиту для безопасного извлечения устройства.

Образец драйвера создает четыре очереди ввода/вывода, три из которых управляются энергопотреблением. Поле *PowerManaged* в структуре *WDF\_IO\_QUEUE\_CONFIG* указывает, является ли очередь управляемой энергопотреблением. Если драйвер не устанавливает это поле, KMDF использует значение по умолчанию, и поэтому создает управляемые энергопотреблением очереди на основе того, что объект устройства играет роль объекта FDO в стеке устройств. Чтобы создать неуправляемую энергопотреблением очередь, драйвер должен явно установить значение этого поля в *WdfFalse*.

Следующая выделенная жирным шрифтом строчка кода регистрирует обратный вызов функции *EvtIoStop* для одной из управляемой энергопотреблением очередей, устанавливая поле *EvtIoStop* структуры *WDF\_IO\_QUEUE\_CONFIG*. Инфраструктура вызывает функцию *EvtIoStop* для управляемой энергопотреблением очереди перед тем, как устройство выходит из рабочего состояния энергопотребления. Эта функция обрабатывает все ожидающие исполнения запросы ввода/вывода таким образом, как это требуется для данного устройства и драйвера.

И в последней выделенной жирным шрифтом строке кода драйвер устанавливает поле *PowerManaged* в структуре *WDF\_IO\_QUEUE\_CONFIG* для его неуправляемой энергопотреблением очереди.

## Пример KMDF: обратные вызовы для входа и выхода из состояния D0

Инфраструктура вызывает функцию обратного вызова *EvtDeviceD0Entry* драйвера сразу же после того, как драйвер переходит в состояние D0, а функцию *EvtDeviceD0Exit* — непосредственно перед тем, как драйвер выходит из этого состояния.

Функция *EvtDeviceD0Entry* выполняет операции, необходимые для перехода устройства в состояние D0. Инфраструктура вызывает эту функцию всякий раз, когда необходимо инициализировать или реинициализировать аппаратное обеспечение устройства. В листинге 7.9 приведен пример функции *EvtDeviceD0Entry* из образца драйвера Osrusbf2.

**Листинг 7.9. Функция обратного вызова *EvtDeviceD0Entry* для простого аппаратного функционального драйвера**

```
NTSTATUS OsrusFxEvtDeviceD0Entry(
    IN WDFDEVICE Device,
    IN WDF_POWER_DEVICE_STATE PreviousState)
```

```

{
    PDEVICE_CONTEXT pDeviceContext;
    NTSTATUS         status;
    PAGED_CODE();
    pDeviceContext = GetDeviceContext(Device);
    status = WdfIoTargetStart(
        WdfUsbTargetPipeCetIoTarget(pDeviceContext->InterruptPipe)
    );
    return status;
}

```

Как можно видеть в листинге 7.9, функция обратного вызова *EvtDeviceDEntry* драйвера Osrusbf2 просто запускает исполнители ввода/вывода драйвера.

Функция обратного вызова *EvtDeviceD0Exit* выполняет операции, требуемые для выхода устройства из состояния D0, например, сохранение состояния аппаратного обеспечения. При выполнении функции *EvtDeviceD0Exit* устройство все еще находится в состоянии D0, поэтому драйвер может обращаться к аппаратному обеспечению. Функция обратного вызова *EvtDeviceD0Exit* драйвера Osrusbf2 показана в листинге 7.10.

**Листинг 7.10. Функция обратного вызова *EvtDeviceD0Entry* для простого аппаратного функционального драйвера**

```

NTSTATUS OsrfxEvtDeviceD0Exit(
    IN WDFDEVICE Device,
    IN WDF_POWER_DEVICE_STATE TargetState)
{
    PDEVICE_CONTEXT pDeviceContext;
    PAGED_CODE();
    pDeviceContext = GetDeviceContext(Device);
    WdfIoTargetStop(WdfUsbTargetPipeCetIoTarget(
        (pDeviceContext->InterruptPipe),
        WdfIoTargetCancelSentIo));
    return STATUS_SUCCESS;
}

```

Функция обратного вызова *EvtDeviceD0Exit* образца драйвера просто отменяет действия, выполненные функцией *EvtDeviceD0Entry*. Таким образом, она останавливает исполнителей ввода/вывода и возвращает *STATUS\_SUCCESS*.

## Действия инфраструктуры для простого аппаратного функционального драйвера

Так же, как и в примерах с чисто программными драйверами, инфраструктура реализует почти всю поддержку для Plug and Play и управления энергопотреблением для только что описанных аппаратных драйверов. Так эти драйверы являются функциональными драйверами, инфраструктура автоматически создает и управляет машинами состояний Plug and Play, управления энергопотреблением и политики энергопотребления. В драйвер нужно добавить только код для управления аппаратным обеспечением устройства.

Инфраструктура вызывает функцию обратного вызова драйвера для подготовки аппаратного обеспечения после обнаружения менеджером PnP устройства, поддерживаемого драйве-

ром, и после возвращения управления функцией обратного вызова драйвера, добавляющей оборудование.

- ◆ Как для UMDF, так и для KMDF драйверов эта функция выполняет инициализацию устройства, требуемую для перехода устройства в состояние D0.

Функции обратного вызова для подготовки аппаратного обеспечения для драйверов USB, таких как, например, образцы драйверов Fx2\_Driver и Osrusbf2, должны вызывать методы, возвращающие информацию об устройстве, и также должны конфигурировать интерфейсы на устройстве.

- ◆ Для драйверов KMDF параметры функции обратного вызова для подготовки аппаратного обеспечения включают дескриптор аппаратных ресурсов, назначенных драйверу.

Драйвер, управляющий ресурсами устройства, для обращения к спискам ресурсов и для манипулирования ими может использовать вспомогательные функции KMDF.

Непосредственно перед тем, как устройство переходит в состояние энергопотребления D0 (рабочий режим), инфраструктура вызывает функцию обратного вызова драйвера для перехода в состояние D0. Одним из параметров этой функции, как для UMDF, так и для KMDF драйверов является состояние энергопотребления, из которого устройство выполняет этот переход. Драйверы обычно игнорируют это значение и инициализируют устройство одинаковым образом, независимо от предыдущего состояния энергопотребления. Значением этого параметра может быть одна из следующих констант перечисления `WDF_POWER_DEVICE_STATE`:

- ◆ `WdfPowerDeviceUnspecified`;
- ◆ `WdfPowerDeviceD0`;
- ◆ `WdfPowerDeviceD1`;
- ◆ `WdfPowerDeviceD1`;
- ◆ `WdfPowerDeviceD3`;
- ◆ `WdfPowerDeviceD3Final`;
- ◆ `WdfPowerDevicePrepareForHibernation`.

При первоначальном включении устройства инфраструктура передает значение `WdfPowerDeviceUnspecified`.

Инфраструктура также передает состояние энергопотребления устройства обратному вызову функции для выхода из состояния D0. В этом случае устройство указывает состояние энергопотребления, в которое оно намеревается перейти после выхода из состояния D0. Два из возможных целевых состояний могут быть вам незнакомы:

- ◆ `WdfPowerDeviceD3Final`;
- ◆ `WdfPowerDevicePrepareForHibernation` (только KMDF).

Инфраструктура передает значение `WdfPowerDeviceD3Final` в качестве целевого состояния энергопотребления устройства, при переходе в состояние D3 в процессе выключения системы или удаления устройства. В таком случае драйвер должен выполнить все требующиеся специфические операции для подготовки к выключению, например, сохранить информацию о состоянии устройства на диск или другой энергонезависимый носитель.

Перед выходом устройства из состояния D0 инфраструктура останавливает все управляемые энергопотреблением очереди ввода/вывода, ассоциированные с этим устройством. После

возвращения устройства в состояние D0 инфраструктура возобновляет работу управляемых энергопотреблением очередей ввода/вывода устройства.

## **KMDF, устройства хранение и гибернация**

Драйверы KMDF для определенных устройств хранения данных могут получать значение `WdfPowerDevicePrepareForHibernation` в качестве целевого состояния, если устройство находится в пути гибернации и система осуществляет подготовку к переходу в режим гибернации. Путь гибернации включает устройство, на котором система сохраняет файл гибернации, и все другие устройства вдоль пути от корня к этому устройству, которые требуются для обеспечения подачи питания устройству.

Инфраструктура передает значение `WdfPowerDevicePrepareForHibernation` только тогда, когда драйвер и вызвал метод `WdfDeviceSetSpecialFileSupport`, и получил извещение о том, что он находится в пути гибернации, но и в этом случае только если целевым состоянием энергопотребления системы является состояние S4.

Когда функция обратного вызова `EvtDeviceD0Exit` драйвера KMDF вызывается со значением `WdfPowerDevicePrepareForHibernation` для целевого состояния, драйвер должен подготовить устройство к переходу в состояние гибернации, выполнив все необходимые операции для приведения устройства в состояние D3, за исключением отключения питания. Сюда входит сохранение любой информации о состоянии устройства, требуемой драйверу для возвращения устройства в состояние D0 после выхода системы из гибернации. Драйвер не должен отключать питание устройства. Система использует устройство при сохранении файла гибернации на диск непосредственно перед переходом в состояние S4.

## **Продвинутые возможности управления энергопотреблением для драйверов KMDF**

Как видно из примеров, приведенных в предыдущем разделе, даже для драйверов аппаратных устройств инфраструктура выполняет большую часть работы по реализации Plug and Play и управления энергопотреблением. В этом разделе рассматривается расширение базовых возможностей драйвера добавлением поддержки для двух продвинутых возможностей KMDF.

- ◆ Поддержка простого устройства.

Драйвер может перевести свое устройство в режим пониженного энергопотребления при простое устройства, когда система находится в рабочем состоянии (S0).

- ◆ Поддержка пробуждения устройства.

Некоторые устройства могут перевести самих себя, а возможно, и всю систему, из состояния пониженного энергопотребления в рабочее состояние полного энергопотребления. Для предоставления поддержки пробуждения должным образом требуются специальные возможности, как и для аппаратного обеспечения, так и для драйвера.

## **Поддержка бездействия устройства в режиме низкого энергопотребления для драйверов KMDF**

При многих обстоятельствах перевод устройства в режим пониженного энергопотребления, когда оно не выполняет никакой работы, т. е. приставляет, но когда система находится в состоянии S0, имеет значительные достоинства:

- ◆ уменьшенное потребление электроэнергии;
- ◆ понижение тепловыделения и шума.

Если аппаратное обеспечение устройства поддерживает переход в режим пониженного энергопотребления при простое устройства, для его использования также необходима поддержка драйвером. Для добавления поддержки энергопотребления бездействия в драйвер KMDF требуется лишь несколько других обратных вызовов в дополнение к обратным вызовам, требуемым для предоставления базовой поддержки Plug and Play.

## Настройки и управление бездействием в драйверах KMDF

Для настройки поддержки бездействия драйвер устанавливает характеристики бездействия в структуре `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS` в своей функции `EvtDriverDeviceAdd` или `EvtDevicePrepareHardware`.

После создания драйвером объекта устройства драйвер инициализирует эту структуру с помощью макроса `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT`. Этот макрос принимает следующие два аргумента:

- ◆ указатель на структуру `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`, которую требуется инициализировать;
- ◆ значение перечисления, указывающее, поддерживает ли драйвер бездействие и поддерживают ли устройство и драйвер пробуждение, когда система находится в состоянии S0.

Список допустимых значений этого параметра приведен в строке `IdleCaps` в табл. 7.5. Драйвер, поддерживающий бездействие, но не поддерживающий пробуждение из состояния S0, должен указывать `IdleCannotWakeFromS0`. Драйвер для устройства USB, поддерживающего выборочную приостановку, должен указывать `IdleUsbSelectiveSuspend` в вызове макроса инициализации.

Если драйвер указывает значения `IdleUsbSelectiveSuspend` или `IdleCanWakeFromS0`, инфраструктура использует зарегистрированные возможности энергопотребления устройства в качестве значения по умолчанию перечислителя `DxState`.

Если же драйвер указывает значение `IdleCannotWakeFromS0`, инфраструктура устанавливает `PowerDeviceD3` в качестве значения по умолчанию перечислителя `DxState`.

После вызова макроса драйвер может также установить значения других полей в структуре `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`. Эти поля перечислены в табл. 7.5.

После инициализации структуры `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS` драйвер вызывает метод `WdfDeviceAssignS0IdleSettings`, передавая ему в параметрах дескриптор объекта устройства и указатель на инициализированную структуру `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`.

Как было уже упомянуто ранее, драйвер может вызывать метод `WdfDeviceAssignS0IdleSettings` в любое время после создания объекта устройства. Хотя большинство драйверов вызывают этот метод из функции обратного вызова `EvtDriverDeviceAdd`, это может быть не всегда возможным или даже нежелательным. Если драйвер поддерживает несколько устройств или версий устройства, драйвер может не знать, способно ли устройство поддерживать пробуждение из состояния S0 до тех пор, пока он не опросит его аппаратное обеспечение. Такие драйверы могут отложить вызов метода `WdfDeviceAssignS0IdleSettings` до вызова функции `EvtDevicePrepareHardware`. Драйвер должен указывать, поддерживает ли устройство пробуждение из состояния S0, при первом вызове метода `WdfDeviceAssignS0IdleSettings`. Инфраструктура не признает изменений в настройках поддержки пробуждения из состояния S0 в последующих вызовах метода `WdfDeviceAssignS0IdleSettings`.

Таблица 7.5. Настройки бездействия устройства KMDF

Имя поля	Описание	Допустимые значения
Enabled	Указывает, разрешается ли переход устройства в режим пониженного энергопотребления при простое	WdfTrue WdfFalse WdfDefault (разрешено, если только явно не запрещено пользователем, имеющим привилегии администратора)
IdleCaps	Указывает, поддерживает ли драйвер бездействие и поддерживают ли устройство и драйвер пробуждение, когда система находится в состоянии S0.  Для драйверов USB указывает, поддерживает ли устройство выборочную простоянку работы USB. Драйверам USB запрещается указывать IdleCanWakeFromS0	IdleCannotWakeFromS0 IdleCanWakeFromS0 IdleUsbSelectiveSuspend
DxState	Указывает состояние энергопотребления, в которое инфраструктура переводит простоявющее устройство	PowerDeviceD0 PowerDeviceD1 PowerDeviceD2  PowerDeviceDB (значение по умолчанию, если значение перечислителя IdleCaps установлено в IdleCannotWakeFromS0)
IdleTimeout	Период времени в миллисекундах, на протяжении которого устройство должно не получать запросов ввода/вывода перед тем, как инфраструктура начинает рассматривать устройство простоявшим	ULONG или IdleTimeoutDefaultValue (текущее состояние — 5000 миллисекунд или 5 секунд)
UserControlOfIdleSettings	Указывает, предоставляет ли инфраструктура страницу свойств в Диспетчере устройств, чтобы позволить администраторам управлять политикой бездействия устройства	IdleDoNotAllowUserControl IdleAllowUserControl

### Примечание

Поддерживает или нет устройство пробуждение из состояния S0, зависит от возможностей как устройства, так и системного разъема, через который оно подключено. Поэтому вызов метода `WdfDeviceAssignS0IdleSettings` с параметром `IdleCanWakeFromS0` может завершиться неудачей с возвращением значения `STATUS_POWER_STATE_INVALID`, если конфигурация аппаратного обеспечения не поддерживает пробуждение из состояния S0. Необходимо обеспечить, чтобы эта ошибка не помешала загрузке драйвера. Если какая-либо функция обратного вызова для инициализации, например `EvtDriverDeviceAdd` или `EvtDevicePrepareHardware`, возвращает это значение инфраструктуре, инфраструктура отключает устройство.

В любое время после первоначального вызова метода `WdfDeviceAssignS0IdleSettings` драйвер может изменить значение длительности простоя устройства для его перевода в состоя-

ние пониженного энергопотребления, в которое устройство переводится при простое, а также данные о том, разрешена ли поддержка перевода устройства в состояние пониженного энергопотребления при простое. Чтобы изменить одну или несколько из этих установок, драйвер просто инициализирует другую структуру `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS` таким же образом, как было описано раньше, и вызывает метод `WdfDeviceAssignS0IdleSettings` опять.

Иногда устройство нельзя переводить в состояние пониженного энергопотребления, даже если к нему и не было запросов ввода/вывода в течение указанного периода простоя перед переходом. В таких случаях драйвер может запретить инфраструктуре переводить простоявшее устройство в состояние пониженного энергопотребления с помощью метода `WdfDeviceStopIdle`. Для восстановления способности устройства переходить в состояние пониженного энергопотребления при простое применяется метод `WdfDeviceResumeIdle`.

Например, образец драйвера `Serial` не переводит свое простоявшее устройство в состояние пониженного энергопотребления, если имеется открытый дескриптор. Этот драйвер вызывает метод `WdfDeviceStopIdle`, когда он получает запрос на открытие дескриптора, и метод `WdfDeviceResumeIdle`, когда получает запрос на закрытие. То же самое справедливо и для драйверов USB. Но это поведение неприменимо для многих других драйверов, т. к. большинство драйверов всегда имеют открытый дескриптор.

Эти два метода управляют подсчетом ссылок на устройство, поэтому, если драйвер вызывает метод `WdfDeviceStopIdle` несколько раз, для восстановления способности устройства переходить в состояние пониженного энергопотребления при простое, драйвер должен вызвать метод `WdfDeviceResumeIdle` такое же количество раз.

Если при вызове метода `WdfDeviceStopIdle` устройство находится в состоянии пониженного энергопотребления, инфраструктура возвращает его в состояние D0. Если при вызове этого метода устройство находится в состоянии D0, то инфраструктура не перезапускает устройство, а только перезапускает таймер периода простоя.

Метод `WdfDeviceStopIdle` не предотвращает переход устройства в режим пониженного энергопотребления, если система перешла в состояние Sx. Он может делать это только тогда, когда система находится в состоянии S0.

### **Запрещение перехода в состояние Dx и взаимоблокировки**

Методу `WdfDeviceStopIdle` передается булев параметр `WaitForD0`, который указывает, когда метод должен возвратить управление — немедленно по его исполнению или же только после того, как устройство возвратится в состояние D0. Если в этом параметре передается значение `TRUE`, то операция перехода в состояние D0 выполняется практически одновременно, что может вызвать взаимоблокировку. Причины этому следующие.

Инфраструктура последовательно выполняет вызовы большинства функций обратного вызова для Plug and Play и управление энергопотреблением, т. е. она вызывает их по одной и ожидает завершения выполнения текущей функции перед тем, как вызывать следующую. Если драйвер вызовет метод `WdfDeviceStopIdle` со значением параметра `WaitForD0` установленным в `TRUE` из функции обратного вызова для Plug and Play или управления энергопотреблением, инфраструктура не сможет вызывать другие функции, которые требуются для возвращения устройства в состояние D0, до тех пор, пока вызвавшая его функция не возвратит управление — чего она никогда не сделает. Поэтому необходимо соблюдать осторожность и вызывать этот метод только из функций, которые вы заранее знаете, не принимают участия в приведение устройства в рабочее состояние энергопотребления.

Не вызывайте метод `WdfDeviceStopIdle` со значением параметра `WaitForD0`, установленным в `TRUE` из следующих функций:

- ◆ любой функции обратного вызова для событий Plug and Play и управления энергопотреблением;
- ◆ любой функции обратного вызова для событий ввода/вывода для управляемой энергопотреблением очереди или любого кода, который вызывается из такой функции обратного вызова.

Если значение параметра `WaitForD0` установлено в `FALSE`, то проблема взаимоблокировки не возникает, т. к. в этом случае метод возвращает управление немедленно и не блокирует вызовы инфраструктурой функций обратного вызова для событий Plug and Play и управления энергопотреблением.

Дополнительная информация о том, как и когда инфраструктура последовательно исполняет функции обратного вызова, приводится в главе 10.

Инфраструктура интегрирует свою обработку перехода устройства в режим пониженного энергопотребления при простое с другой деятельностью драйвера по обработке событий Plug and Play и управлению энергопотреблением. Инфраструктура переводит устройство в состояние энергопотребления, которое драйвер указал в своем последнем вызове метода `WdfDeviceAssignS0IdleSettings`, при удовлетворении всех следующих условий:

- ◆ разрешен переход в состояние пониженного энергопотребления при простое;
- ◆ драйвер не деактивировал переход в состояние пониженного энергопотребления при простое, вызвав метод `WdfDeviceStopIdle`;
- ◆ период ожидания для перехода в режим пониженного энергопотребления истек;
- ◆ на устройстве нет активных запросов ввода/вывода.

Инфраструктура возвращает устройство в состояние D0 всякий раз, когда происходит одно из следующих событий:

- ◆ прибывает новый запрос ввода/вывода на одну из управляемых энергопотреблением очередей устройства;
- ◆ драйвер вызывает метод `WdfDeviceStopIdle`;
- ◆ драйвер запрещает переход в режим пониженного энергопотребления при простое, вызывая метод `WdfDeviceAssignS0IdleSettings` и передавая ему в качестве параметра структуру `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`, в которой значение члена `Enabled` установлено в `WdfFalse`;
- ◆ система переходит в состояние энергопотребления, несовместимое с состоянием энергопотребления устройства.

## **Выбор периода простоя и состояний пониженного энергопотребления в драйверах KMDF**

Здесь следует сказать несколько слов о том, как выбирать длительность периода простоя перед переходом в состояние пониженного энергопотребления и состояния энергопотребления устройств. В общем случае разница в задержках при возвращении устройства в состояние D0 из состояний D1, D2 и D3 составляет несколько миллисекунд. Но этому есть несколько исключений. Например, для видеодисплеев разница в задержках может составлять несколько секунд. Но почти для всех других устройств разница между возвращением из со-

стояния D1 и из состояния D3 настолько небольшая, что практически незаметна для пользователя. Соответственно, принимая во внимание большую экономию электроэнергии при простоянии устройства в полностью отключенном состоянии, инфраструктура по умолчанию переводит простоявшие устройства в режим D3.

Раньше некоторые поставщики не решались снабжать свои устройства поддержкой перехода в состояние пониженного энергопотребления при простое, т. к. они думали, что пользователи принимали задержку при возвращении из этого режима за пониженную производительность устройства. Но такой подход не позволяет этим устройствам извлекать выгоду пониженного потребления электроэнергии, тепловыделения и шума, которую может предоставить поддержка перехода в состояние пониженного энергопотребления при простое. Исследования, выполненные компанией Microsoft, показывают, что пользователи воспринимают почти любую задержку приемлемой, если она происходит только при выходе системы (или, возможно, устройства) из полностью отключенного состояния. Драйвер KMDF может предотвратить свое устройство от преждевременного перехода в состояние пониженного энергопотребления путем увеличения значения члена `IdleTimeout` перечисления `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`, а также вызывая метод `WdfDeviceAssignS0IdleSettings` каждый раз, когда устройство выполняет работу.

## Поддержка пробуждения устройства для драйверов KMDF

Состояния энергопотребления системы, при которых устройство может инициировать сигнал пробуждения, зависят как от конструкции устройства, так и от конструкции системы. Наиболее распространенными моделями применения сигнала пробуждения являются следующие три.

- ◆ **Пробуждение из состояния S0: активирование сигнала пробуждения из состояния S0.**

Если система пребывает в состоянии S0 и устройство бездействует, внешний сигнал заставляет устройство активировать сигнал пробуждения, что, в свою очередь, заставляет инфраструктуру и драйвер возвратить устройство в рабочее состояние. Таким внешним сигналом может быть щелчок мыши или подключение сетевого кабеля к сетевому адаптеру.

- ◆ **Пробуждение из состояния Sx: активирование сигнала пробуждения из состояния S1, S2, S3 или S4.**

Если система пребывает в режиме ожидания, драйвер может активировать сигнал пробуждения, чтобы возвратить систему в рабочее состояние.

Сигнал для активирования сигнала пробуждения прибывает извне, но условия, вызывающие сигнал пробуждения, часто разные. Например, вы, скорее всего, не хотели, чтобы компьютер вышел из текущего режима при подключении сетевого кабеля, но совсем другое дело, если из сети прибывает специальный пакет. Поэтому инфраструктура поддерживает разные функции обратного вызова для подготовки устройства для пробуждения из состояния S0 и из состояний Sx.

- ◆ **Пробуждение из состояния S5: активирование сигнала пробуждения из состояния S5.**

Некоторые устройства могут активировать сигнал пробуждения из состояния S5, что вызывает приведение компьютера в рабочее состояние из полностью выключенного состояния.

Эта возможность часто применяется для удаленного управления по сети. Но эта возможность вне рамок реализации с помощью драйверов и Windows, т. к. для этого требуется решение BIOS. Пробуждение системы из состояния S5 не считается пробуждением с точки зрения программного обеспечения, т. к. компьютер не находится в состоянии пониженного энергопотребления, а полностью выключен. Поэтому реализация пробуждения из состояния S5 лежит в области разработки аппаратуры, в контексте BIOS, а не контексте операционной системы и драйверов.

Пожалуй, наиболее распространенным видом пробуждения является пробуждение из состояния Sx. Рассмотрим следующий пример. Когда система пребывает в состоянии S3, с соответствующим пребыванием в состояния Dx устройств, подключенных к системе, по сети прибывает магический пакет. В результате получения этого пакета, аппаратное обеспечение сетевой платы (которая перед тем, как перейти в состояние пониженного энергопотребления, была должным образом сконфигурирована) активирует событие пробуждения (т. е. PME# на шине PCI), что вызывает переход системы в состояние S0. В результате система и подсоединенные к ней устройства пробуждаются и возвращаются в полностью рабочее состояние.

В отличие от пробуждения из состояния Sx, пробуждение из состояния S0 связано с поддержкой для перехода в состояние пониженного энергопотребления при бездействии. Когда устройство бездействует, оно переходит в состояние пониженного энергопотребления, в то время как система остается в состоянии S0. Пробуждение из состояния S0 просто означает, что устройство может активировать сигнал пробуждения со своего состояния пониженного энергопотребления. Между поддержкой пробуждения и поддержкой перехода в состояние пониженного энергопотребления существуют следующие взаимоотношения:

- ◆ устройства, поддерживающие переход в состояние пониженного энергопотребления при простое, когда система пребывает в состоянии S0, не обязательно поддерживают пробуждение из состояния S0;
- ◆ устройства, поддерживающие пробуждение из состояния S0, также неявно поддерживают переход в режим пониженного энергопотребления при простое;
- ◆ устройства, поддерживающие пробуждение из состояния Sx, не обязательно поддерживают переход в состояние пониженного энергопотребления при простое, но могут делать это.

Настройки пробуждения, которые должен поддерживать драйвер, зависят от сценариев, выбранных для поддержки устройством. Например, мышь, сетевая плата и последовательный порт могут поддерживать пробуждение следующим образом.

- ◆ Мышь активирует сигнал пробуждения, когда пользователь сдвинет ее или щелкнет кнопкой. Это может произойти в состоянии S0 или Sx, в зависимости от системной политики энергопотребления.
- ◆ Сетевая плата переходит в режим пониженного энергопотребления при извлечении сетевого кабеля, и активирует сигнал пробуждения в состоянии S0, когда в нее вставляется сетевой кабель. Сетевая плата также может активировать пробуждение в состоянии Sx, когда прибывает магический пакет. Если система оборудована специальной BIOS, сетевая плата может активировать пробуждение из состояния S5, когда специальное приложение управления запускает систему удаленно для обслуживания.
- ◆ Драйвер не может знать, какой тип устройства подключен через последовательный порт. Поэтому всегда, когда система пребывает в состоянии S0 и открыт дескриптор последовательного порта, последовательный порт должен быть в состоянии D0. Последователь-

ный порт может активировать пробуждение из состояния Sx при активировании вывода *Ring Indicate*, что указывает, что к порту подключен модем, и на соответствующий телефон поступает сигнал вызова.

В следующих разделах описываются способы реализации пробуждения как из состояния Sx, так и из состояния S0 в драйвере KMDF.

## Реализация пробуждения из состояния Sx в драйверах KMDF

Для разрешения поддержки пробуждения из состояния Sx драйвер применяет следующие две структуры.

- ◆ WDF\_POWER\_POLICY\_EVENT\_CALLBACKS — структура, которая содержит указатели на следующие функции обратного вызова для событий политики энергопотребления устройства:
  - EvtDeviceArmWakeFromS0;
  - EvtDeviceDisarmWakeFromS0;
  - EvtDeviceWakeFromS0Triggered;
  - EvtDeviceArmWakeFromSx;
  - EvtDeviceDisarmWakeFromSx;
  - EvtDeviceWakeFromSxTriggered.

Эта структура вставляется в структуру DEVICE\_INIT перед созданием объекта устройства.

- ◆ WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS — структура, которая содержит конфигурационные данные для пробуждения устройства, включая состояние энергопотребления, из которого устройство пробуждает систему и управление пробуждением пользователем. Драйвер заполняет эту структуру, вызывая метод WdfDeviceAssignSxWakeSettings после создания объекта устройства.

### Функции обратного вызова по событиям политики энергопотребления для пробуждения из состояния Sx

Драйвер инициализирует структуру WDF\_POWER\_POLICY\_EVENT\_CALLBACKS в своей функции обратного вызова EvtDriverDeviceAdd. Эта структура используется в качестве ввода для структуры WDFDEVICE\_INIT, и ее необходимо организовать до создания объекта устройства драйвером.

Драйвер инициализирует эту структуру с помощью макрояза WDF\_POWER\_POLICY\_EVENT\_CALLBACKS\_INIT, который проставляет в ней указатели на свои функции обратного вызова EvtDeviceArmWakeFromSx, EvtDeviceDisarmWakeFromSx и EvtDeviceWakeFromSxTriggered в одноименных полях.

Инфраструктура запрашивает драйвер разрешить своему устройству пробуждение из состояния Sx и подготовить его к этой операции, вызывая для этого функцию драйвера EvtDeviceArmWakeFromSx. В этой функции драйвер исполняет специфичные для устройства операции для разрешения пробуждения и подготовки устройства к его выполнению. Если драйверу не требуется выполнять никаких специфичных для устройства задач по приведению устройства в готовность для пробуждения (например, переконфигурировать сигнал внутреннего прерывания), эта функция не вызывается.

Функция обратного вызова EvtDeviceDisarmWakeFromSx отменяет все действия, выполненные функцией EvtDeviceArmWakeFromSx. Инфраструктура вызывает ее с тем, чтобы драйвер отме-

нил готовность своего устройства пробуждаться из состояния Sx. Как и в случае с функцией *EvtDeviceArmWakeFromSx*, если для отмены готовности устройства пробуждаться не требуются никакие специфичные для устройства действия, драйвер не регистрирует эту функцию для вызова.

Когда устройство драйвера активирует сигнал пробуждения, инфраструктура вызывает функцию обратного вызова *EvtDeviceWakeFromSxTriggered*. Так как всеми аспектами пробуждения системы занимается инфраструктура, то вызов этой функции делается в чисто информационных целях. Соответственно, большинство драйверов не регистрируют эту функцию для вызова.

### Примечание

Инфраструктура вызывает функции *EvtDeviceWakeFromSxTriggered* и *EvtDeviceWakeFromS0-Triggered*, только если материнская плата и системная BIOS реализованы должным образом и работают безошибочно, что иногда не совсем так. Если для правильной работы драйвера ему необходимо знать, когда устройство активирует сигнал пробуждения, устройство должно само предоставить эту информацию, а функции обратного вызова семейства *EvtDeviceDisarmWakeXxx* должны считывать эту информацию с устройства.

После того как драйвер заполнит все поля, относящиеся к его реализации, он регистрирует функции обратного вызова в структуре *WDFDEVICE\_INIT*, вызывая для этого метод *WdfDeviceInitSetPowerPolicyEventCallbacks*. После этого драйвер может выполнить дополнительные операции по инициализации или создать объект устройства.

### Настройки политики энергопотребления для пробуждения из состояния Sx

После создания объекта устройства драйвер может также инициализировать структуру *WDF\_DEVICE\_POWER\_POLICY\_WAKE\_SETTINGS*. Эта структура содержит информацию о поддерживаемой политике пробуждения устройства из состояния Sx и поставляется в качестве параметра метода *WdfDeviceAssignSxWakeSettings*, чтобы зарегистрировать эту поддержку в инфраструктуре. Поля структуры перечислены в табл. 7.6.

**Таблица 7.6. Настройки пробуждения устройства из состояния Sx**

Имя поля	Описание	Допустимые значения
Enabled	Указывает, может ли устройство пробуждать систему из состояния пониженного энергопотребления	WdfTrue WdfFalse WdfDefault (разрешено, если только явно не запрещено пользователем, имеющим привилегии администратора)
DxState	Указывает состояние энергопотребления устройства, в которое инфраструктура переводит устройство, когда система переходит в состояние пониженного энергопотребления, из которого возможно пробуждения	PowerDeviceD1 PowerDeviceD2 PowerDeviceD3
UserControlOfWakeSettings	Указывает, предоставляет ли инфраструктура страницу свойств в Диспетчере устройств, чтобы позволить администраторам управлять возможностью устройства пробуждать систему	WakeDoNotAllowUserControl WakeAllowUserControl

Структуру `WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS` драйвер инициализирует с помощью макрока `WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT`. После этого он устанавливает значения для своего устройства в соответствующие поля этой структуры.

Поле `DxState` драйвер заполняет значением состояния энергопотребления устройства, в которое инфраструктура должна перевести устройство, когда оно подготовлено к пробуждению из состояния `Sx`. По умолчанию инфраструктура использует значение, предоставленное в данных о возможностях энергопотребления устройства.

Драйвер также заполняет поле `UserControlOfWakeSettings`, указывая, могут ли пользователи, имеющие надлежащие привилегии, управлять политикой пробуждения устройства. Если это поле имеет значение `WakeAllowUserControl`, то инфраструктура автоматически создает для драйвера страницу свойств в Диспетчере устройств, с помощью которой пользователи, имеющие привилегии администратора, могут разрешить или запретить пробуждение устройства. Если устройство поддерживает как возможность пробуждения, так и возможность перехода в режим пониженного энергопотребления при простое и обе эти возможности позволяют пользователям управлять своими политиками, то по умолчанию они выводятся вместе на одной странице свойств в Диспетчере устройств. Эта страница свойств модифицирует значения реестра `IdleInWorkingState` и `WakeFromSleepState`, которые хранятся в подключе `Parameters\Wdf` для данного узла `devnode`. Пользователям и драйверам запрещается прямой доступ к этим значениям реестра.

Драйвер может запретить управление политикой безопасности пользователем, установив значение этого поля в `WakeDoNotAllowUserControl`. Но в большинстве случаев драйверы должны разрешать пользователям управлять политикой пробуждения, т. к. при некоторых вариантах конфигурации аппаратного обеспечения поддержка возможности пробуждения оставляет желать лучшего.

После полной инициализации структуры `WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS` драйвер вызывает метод `WdfDeviceAssignSxWakeSettings`, передавая ему в параметрах указатель на эту структуру и дескриптор объекта `WDFDEVICE`.

## Реализация пробуждения из состояния S0 в драйверах KMDF

Реализация поддержки пробуждения устройства из состояния `S0` сходна с реализацией поддержки пробуждения устройства из состояния `Sx`. Инициализация обычно выполняется в функции драйвера `EvtDriverDeviceAdd`. Но т. к. поддержка пробуждения из состояния `S0` связана с поддержкой перехода устройства в режим пониженного энергопотребления при простое, то драйвер должен реализовать поддержку режима пониженного энергопотребления при бездействии и указывать, что он поддерживает пробуждение из состояния `S0` при разрешении поддержки перехода в режим пониженного энергопотребления при простое.

Драйвер указывает, что он поддерживает пробуждение из состояния `Sx` с помощью следующих структур:

- ◆ `WDF_POWER_POLICY_EVENT_CALLBACKS`;
- ◆ `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS`.

Структура `WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS` не используется для конфигурирования поддержки пробуждения из состояния `S0`.

## Функции обратного вызова по событиям политики энергопотребления для пробуждения из состояния S0

Драйвер инициализирует структуру `WDF_POWER_POLICY_EVENT_CALLBACKS` в своей функции обратного вызова `EvtDriverDeviceAdd`. Эта структура используется в качестве ввода для струк-

туры `WDFDEVICE_INIT`, и ее необходимо организовать до создания объекта устройства драйвером.

Как было описано ранее, драйвер инициализирует структуру с помощью макроса `WDF_POWER_POLICY_EVENT_CALLBACKS_INIT`. Он проставляет указатели на функции обратного вызова `EvtDeviceArmWakeFromS0`, `EvtDeviceDisarmWakeFromS0` и `EvtDeviceWakeFromS0Triggered` в одноименных полях структуры.

Функции обратного вызова `EvtDeviceArmWakeFromS0` и `EvtDeviceDisarmWakeFromS0` исполняют специфичные для устройства операции для приведения устройства в готовность к пробуждению (например, для пробуждения сетевой платы при подключении сетевого кабеля), когда система пребывает в состоянии S0, и отмены этой готовности. Эти функции необходимы, только если для драйвера и устройства требуется выполнение следующих действий.

- ◆ Если для подготовки устройства к пробуждению как из состояния S0, так и из состояния Sx требуется выполнение одинаковых операций, драйвер может указать одни и те же функции обратного вызова в полях `EvtDeviceArmWakeFromS0` и `EvtDeviceArmWakeFromSx` структуры `WDF_POWER_POLICY_EVENT_CALLBACKS`.
- ◆ Если для подготовки устройства для пробуждения системы из состояния S0 не требуется выполнение специфичных для устройства операций, то функция обратного вызова `EvtDeviceArmWakeFromS0` не является необходимой. То же самое относится и к функции обратного вызова `EvtDeviceDisarmWakeFromS0`.

Так же, как и функция `EvtDeviceWakeFromSxTriggered`, функция `EvtDeviceWakeFromS0Triggered` вызывается в информационных целях, и с ее вызовом необходимо соблюдать те же предосторожности, которые изложены в предыдущем разделе.

Когда структура `WDF_POWER_POLICY_EVENT_CALLBACKS` полностью инициализирована, драйвер регистрирует функции обратного вызова в структуре `WDFDEVICE_INIT`, вызывая для этого метод `WdfDeviceInitSetPowerPolicyEventCallbacks`.

Если драйвер поддерживает пробуждение как из состояния S0, так и из состояния Sx, то перед тем, как вызывать метод `WdfDeviceInitSetPowerPolicyEventCallbacks`, он заполняет структуру `WDF_POWER_POLICY_EVENT_CALLBACKS` значениями для функций обратных вызовов, требуемых для поддержки пробуждения из обоих этих состояний.

### **Настройки политики энергопотребления бездействия для пробуждения из состояния S0**

После создания объекта устройства драйвер инициализирует структуру `WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS`, вызывая для этого макрос `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT`.

Как обсуждалось ранее, этот макрос принимает аргумент, который указывает, поддерживают ли устройство и драйвер пробуждение из состояния S0. Драйвер, поддерживающий пробуждение из состояния S0, должен указывать `IdleCanWakeFromS0` или `IdleUsbSelectiveSuspend` для устройств USB при вызове этого макроса. Драйвер потом устанавливает другие поля в этой структуре и вызывает метод `WdfDeviceAssignSOIdleSettings`, как было описано в разд. "Поддержка бездействия устройства в режиме низкого энергопотребления для драйверов KMDF" ранее в этой главе.

### **Пример KMDF: поддержка бездействия и пробуждения устройств**

В этом примере продолжается рассмотрение аппаратного функционального драйвера из разд. "Пример KMDF: код для Plug and Play и управления энергопотреблением в простом

аппаратном функциональном драйвере" ранее в этой главе. Исходный код для функции обратного вызова `EvtDriverDeviceAdd` драйвера можно найти в листинге 7.8 в том разделе.

Драйвер Osrusbfx2 добавляет поддержку для бездействия устройства и выборочную приостановку USB-устройств. Для этого он инициализирует структуру `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS` и вызывает метод `WdfDeviceAssignS0IdleSettings`. Перед тем как драйвер может инициализировать поддержку простоя и пробуждения, он должен определить, поддерживается ли эти возможности устройством. Чтобы получить эту информацию, драйвер опрашивает устройство с помощью функции обратного выбора `EvtDevicePrepareHardware`, которая приведена в листинге 7.11. Эта функция определена в исходном файле `Driver.c`.

#### Листинг 7.11. Функция обратного вызова `EvtDevicePrepareHardware`

```
NTSTATUS OsrfxEvtDevicePrepareHardware(
    IN WDFDEVICE Device,
    IN WDFCMRESLIST ResourceList,
    IN WDFCMRESLIST ResourceListTranslated)
{
    NTSTATUS status;
    PDEVICE_CONTEXT pDeviceContext;
    WDF_USB_DEVICE_INFORMATION deviceInfo;
    ULONG waitWakeEnable;
    UNREFERENCED_PARAMETER(ResourceList);
    UNREFERENCED_PARAMETER(ResourceListTranslated);

    pDeviceContext = GetDeviceContext (Device);
    // Создается дескриптор устройства USB для
    // взаимодействия с нижележащим стеком USB.
    status = WdfUsbTargetDeviceCreate (Device,
                                      WDF_NO_OBJECT_ATTRIBUTES,
                                      &pDeviceContext->UsbDevice);

    if (!NT_SUCCESS(status)) {
        return status;
    }
    status = SelectInterfaces(Device);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    // Получаем информацию о USB.
    WDF_USB_DEVICE_INFORMATION_INIT(&deviceInfo);
    status = WdfUsbTargetDeviceRetrieveInformation
        (pDeviceContext->UsbDevice, &deviceInfo);
    waitWakeEnable = deviceInfo.Traits &
        WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE;
    // Разрешается пробуждение и период бездействия
    // (если поддерживается устройством).
    if(waitWakeEnable) {
        status = OsrfxSetPowerPolicy(Device);
        if (!NT_SUCCESS(status)) {
            return status;
        }
    }
}
```

```

status = OsrFxConfigContReaderForInterruptEndPoint(pDeviceContext);
return status;
}

```

В листинге 7.11 показан пример, каким образом драйвер может определить, поддерживает ли его устройство пробуждение. В этом примере драйвер создает объект устройства исполнителя USB, после чего вызывает метод `WdfUsbTargetDeviceRetrieveInformation`, чтобы определить возможности устройства и нижележащего в стеке драйвера порта. Информация об этих возможностях возвращается в виде набора битовых флагов в поле `Traits` структуры `WDF_USB_DEVICE_INFORMATION`. Драйвер тестирует значение бита `WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE` и, если этот бит установлен, вызывает внутреннюю вспомогательную функцию `OsrFxSetPowerPolicy`, чтобы разрешить поддержку бездействия и пробуждения.

Исходный код функции `OsrFxSetPowerPolicy` показан в листинге 7.12. Эта функция определена в исходном файле `Device.c`.

#### Листинг 7.12. Инициализация поддержки пробуждения и бездействия в драйвере USB KMDF

```

NTSTATUS OsrFxSetPowerPolicy(
    IN WDFDEVICE Device
)
{
    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS idleSettings;
    WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS wakeSettings;
    NTSTATUS status = STATUS_SUCCESS;
    // Инициализация структуры для политики бездействия.
    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT(&idleSettings,
                                                IdleUsbSelectiveSuspend);
    idleSettings.IdleTimeout = 10000; // 10 секунд
    status = WdfDeviceAssignS0IdleSettings(Device, &idleSettings);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    // Инициализация структуры для политики бездействия и пробуждения.
    WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT(&wakeSettings);
    status = WdfDeviceAssignSxWakeSettings(Device, &wakeSettings);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    return status;
}

```

Функция `OsrFxSetPowerPolicy` вызывается из функции обратного вызова `EvtDevicePrepareHardware` драйвера, чтобы разрешить бездействие и пробуждение устройства USB.

В этом примере драйвер вызывает метод `WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT`, указывая значение `IdleUsbSelectiveSuspend`. Драйвер устанавливает значение `IdleTimeout` в 10 000 миллисекунд (10 секунд) и принимает инфраструктурные установки по умолчанию для значений `DxState` и `UserControlOfIdleSettings`. В результате инфраструктура переводит устройство в состояние D3 при простое. Также создается страницу свойств в Диспетчере устройств, чтобы позволить пользователям с привилегиями администратора разрешать

или запрещать поддержку бездействия устройства. Драйвер потом вызывает метод `WdfDeviceAssignS0IdleSettings`, чтобы разрешить поддержку бездействия и зарегистрировать эти настройки в инфраструктуре.

Для устройств USB, поддерживающих выборочную приостановку, их драйвер подготавливает аппаратное обеспечение устройства к пробуждению. Соответственно, функциональный драйвер должен предоставить функцию обратного вызова `EvtDeviceArmWakeFromS0`, только если требуется выполнение дополнительных специфичных для устройства операций инициализации. Инфраструктура посыпает запрос на выборочную приостановку драйверу шины USB по истечении периода простоя.

## Действия инфраструктуры для поддержки бездействия устройства

Инфраструктура ведет подсчет активности ввода/вывода на всех управляемых энергопотреблением очередях, принадлежащих каждому устройству объекта. Если драйвер поддерживает бездействие для объекта устройства, инфраструктура запускает таймер обратно отсчета периода простоя, когда подсчет запросов ввода/вывода примет нулевое значение. Длительность периода простоя таймера в миллисекундах устанавливается в поле `IdleTimeout`, переданном методу `WdfDeviceAssignS0IdleSettings` в последний раз.

Если в управляемую энергопотреблением очередь устройства прибывает запрос ввода/вывода или драйвер вызывает метод `wdfDeviceStopIdle` до истечения периода `IdleTimeout`, инфраструктура отменяет таймер.

Если же ни одно из этих событий не происходит и период простоя истекает (т. е. таймер принимает нулевое значение), инфраструктура выполняет требуемые действия для перевода устройства из состояния D0 в состояние энергопотребления, указанное драйвером в последнем переданном методу `WdfDeviceAssignS0IdleSettings` поле `DxState`.

Независимо от причины перехода в состояние бездействия, инфраструктура всегда обрабатывает переход из состояния D0 одинаковым образом. Таким образом, когда устройство переходит из состояния D0 в состояние Dx, инфраструктура активирует функции обратного вызова соответственно последовательностям перехода в состояние пониженного энергопотребления, описанных в разд. *"Отключение питания и удаление устройства"* ранее в этой главе.

Инфраструктура автоматически возвращает простоявшее устройство в состояние D0 всякий раз, когда счетчик деятельности ввода/вывода на любой из управляемых энергопотреблением очередей принимает ненулевое значение или когда драйвер вызывает метод `wdfDeviceStopIdle`. И снова, переход в состояние D0 всегда выполняется соответственно последовательностей перехода в рабочее состояние, описанных в разд. *"Перечисление и запуск устройства"* ранее в этой главе.

## Действия инфраструктуры, поддерживающие пробуждение устройства

Если драйвер поддерживает возможность пробуждения своего устройства, инфраструктура вызывает функцию обратного вызова `EvtDeviceArmWakeFromSx` драйвера при переходе системы в состояние Sx, за исключением состояния S5 (полностью выключено), если драйвер

поддерживает пробуждение из нового состояния энергопотребления системы. Например, допустим, что система переходит в состояние S3. Если драйвер поддерживает пробуждение из состояния S3, инфраструктура вызывает функцию драйвера *EvtDeviceArmWakeFromSx*. Но если драйвер поддерживает пробуждение только из состояния S1, инфраструктура не вызывает драйвер для подготовки к сигналу пробуждения.

### **Запросы о состоянии энергопотребления системы**

Перед переводом системы в другое состояние энергопотребления Windows опрашивает устройство, поддерживает ли оно пробуждение системы из этого состояния. Это вызывает вопрос, если устройство не может поддерживать пробуждение из предлагаемого состояния энергопотребления системы, почему тогда инфраструктура разрешает такой переход?

Этому есть две причины.

- Если инфраструктура не позволит переход в данное состояние Sx, а потом не разрешит какому-то другому драйверу переход в другое состояние Sx, система может никогда не перейти в состояние пониженного энергопотребления.
- Инфраструктура не хочет запретить системе переходить в более глубокий режим ожидания только из-за того, что устройство не может вывести ее из этого состояния. Или, иными словами, инфраструктура не рассматривает предоставление возможности устройству пробуждать систему достаточным основанием для перевода системы в более легкий режим ожидания.

(Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation*, Microsoft.)

Приведем объявление функции обратного вызова *EvtDeviceArmWakeFromSx*:

```
NTSTATUS EvtDeviceArmWakeFromSx(WDFDEVICE Device)
```

Инфраструктура вызывает эту функцию, прежде чем предпринять какие-либо действия по переводу устройства в состояние Dx, например, вызов функции *EvtDeviceDOExit*. В функции *EvtDeviceArmWakeFromSx* драйвер приводит устройство в готовность к пробуждению из состояния Sx. Если драйвер не может самостоятельно привести устройство в готовность к пробуждению, функция завершается неудачно и возвращает ошибку, а инфраструктура продолжает перевод устройства в состояние Dx неподготовленным к пробуждению. Если впоследствии система и устройство возвращаются в рабочий режим, а потом опять переводятся в режим ожидания, по умолчанию инфраструктура опять вызывает функцию обратного вызова *EvtDeviceArmWakeFromSx* драйвера. То есть, инфраструктура не помнит, что драйвер был не в силах подготовить устройство к пробуждению при предыдущем переходе в пониженный режим энергопотребления.

Драйвер может отключить повторные запросы приведения устройства в готовность к пробуждению из состояния Sx; для этого функция *EvtDeviceArmWakeFromSx* возвращает значение *WDFSTATUS\_ARM\_FAILED\_DO\_NOT\_RETRY*.

Если устройство активирует пробуждение системы, инфраструктура вызывает функцию *EvtDeviceWakeFromSxTriggered* его драйвера. Так как система выполняет всю необходимую работу по пробуждению системы, эта функция вызывается лишь для того, чтобы держать драйвер в курсе.

Когда система возвращается в состояние S0, инфраструктура вызывает функцию *EvtDeviceDisarmWakeFromSx* драйвера после возвращения устройства в рабочее состояние, т. е. после вызова функции *EvtDeviceDOEntry*. Функция *EvtDeviceDisarmWakeFromSx* отменяет го-

твность устройства к пробуждению, а также отменяет все другие специфичные для устройства операции, выполненные вызовом функции *EvtDeviceArmWakeFromSx*.

Поддержка пробуждения из состояния S0 реализуется почти так же, как и поддержка пробуждения из состояния Sx. Единственная разница состоит в том, что инфраструктура вызывает функции *EvtDeviceArmWakeFromS0* и *EvtDeviceDisarmWakeFromS0* драйвера, когда устройство готово к переходу или выходу из состояния пониженного энергопотребления соответственно. Так же, как и в случае с функциями *EvtDeviceArmWakeFromSx* и *EvtDeviceDisarmWakeFromSx*, эти функции вызываются перед другими функциями обратного вызова драйвера, связанными с входом или выходом из состояния D0.

# ГЛАВА 8

## Поток и диспетчеризация ввода/вывода

Основой любого драйвера является обработка ввода/вывода. Чтобы понимать, как реализовать отказоустойчивый код для обработки ввода/вывода в драйвере, необходимо понимать, каким образом происходит поток запросов ввода/вывода в драйвере, как WDF упрощает работу драйвера по обработке ввода/вывода и что должен делать драйвер для выполнения запроса ввода/вывода.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы</b>	
AMCC5933	%wdk%\Src\Kmdf\AMCC5933
Featured Toaster	%wdk%\Src\Kmdf\Toaster\Func\Featured
Fx2_Driver	%wdk%\Src\Umdf\Usb\Fx2_driver
Nonpnp	%wdk%\Src\Kmdf\Nonpnp
Osrusbf2	%wdk%\Src\Kmdf\Osrusbf2
Pcidrv	%wdk%\Src\Kmdf\Pcidrv
USB\Filter	%wdk%\Src\Umdf\Usb\Filter
WpdWudfSampleDriver	%wdk%\Src\Umdf\Wpd\WpdWudfSampleDriver
<b>Документация WDK</b>	
Handling I/O Requests in Framework-based Drivers <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80613">http://go.microsoft.com/fwlink/?LinkId=80613</a>
<b>Прочее</b>	
I/O Completion/Cancellation Guidelines <sup>2</sup> на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=82321">http://go.microsoft.com/fwlink/?LinkId=82321</a>

<sup>1</sup> Обработка запросов ввода/вывода в драйверах, основанных на инфраструктуре (WDF). — *Пер.*

<sup>2</sup> Руководство по завершению/отмене ввода/вывода. — *Пер.*

## Основные типы запросов ввода/вывода

Основной задачей при разработке драйвера для обработки ввода/вывода является определение типов запросов, которые драйвер должен обрабатывать, типов запросов, которые он должен передавать вниз по стеку устройств, и типов запросов, которые могут обрабатываться инфраструктурой вместо драйвера.

Приложения наиболее часто выдают запросы на создание, закрытие, чтение и запись, и запросы IOCTL.

### Запросы на создание

Приложение обычно открывает файл, каталог или устройство, вызывая функцию Windows CreateFile. При успешном выполнении запроса система возвращает дескриптор файла, с помощью которого приложение может выполнять операции ввода/вывода. Дескриптор файла является специфичным для процесса, создавшего его, но не для потока. Приложение предоставляет дескриптор файла всем последующим запросам ввода/вывода для идентификации получателя запроса.

Всякий раз, когда какое-либо приложение пытается открыть дескриптор файла, менеджер ввода/вывода Windows создает файловый объект и посыпает запрос на создание в стек целевого устройства. Когда WDF получает запрос на создание, она обычно создает файловый объект WDF, который соответствует файловому объекту менеджера ввода/вывода. Несмотря на такое имя, ни файловый объект WDF, ни файловый объект менеджера ввода/вывода не обязательно представляют файл на физическом устройстве. В действительности файловый объект предоставляют способ, с помощью которого драйвер может отслеживать установленный сеанс с приложением, а приложение указывать, какой интерфейс устройства необходимо открыть.

Драйвер может сделать выбор получать и обрабатывать запросы на создание самому, или же он может оставить эту задачу для выполнения инфраструктурой от его имени. Помните, что несколько процессов могут иметь дескрипторы одного и того же файлового объекта, откуда следует, что несколько процессов могут пользоваться одним и тем же файлом одновременно. Если дочерний процесс унаследует дескриптор от своего родительского процесса или если процесс скопирует дескриптор из другого процесса, файл может находиться в совместном использовании несколькими пользователями одновременно. Но при всем этом дескриптор файла представляет лишь один сеанс. Приложение, которое разделяет дескриптор, разделяет свой сеанс с другим приложением.

### Запросы на очистку и закрытие

Когда дескриптор файла больше не нужен приложению, оно обычно вызывает функцию CloseHandle. В ответ менеджер ввода/вывода уменьшает значение счетчика дескрипторов для соответствующего файлового объекта. Когда все дескрипторы файлового объекта закрыты, менеджер ввода/вывода посыпает драйверу запрос на очистку. В ответ на этот запрос драйвер должен отменить все оставшиеся запросы ввода/вывода для данного файлового объекта. Хотя, когда драйвер получает запрос на очистку, открытых дескрипторов больше нет, на файловый объект могут все еще существовать ссылки, такие как, например, вызванные ожидающим исполнения пакетом IRP.

Когда счетчик ссылок на объект примет нулевое значение и все операции ввода/вывода для файла завершены, менеджер ввода/вывода посыпает драйверу запрос на очистку.

### Функция CloseHandle и очистка

Для разработчиков приложений пользовательского режима иногда становится сюрпризом, что результатом вызова функции `CloseHandle` является запрос на очистку, а не на закрытие. Лично мне бы хотелось, чтобы вместо `IRP_MJ_CLOSE` использовалось другое имя, скажем `IRP_MJ_FINALIZE`, которое позволило бы избежать подобных недоразумений.

Когда приложение закрывает дескриптор (при условии, что это единственный неудаленный дескриптор), драйвер получает запрос на очистку, который просто означает, что у файлового объекта больше нет клиентов. Поэтому все операции ввода/вывода для данного файлового объекта необходимо отменить. Таким образом, очистка просто служит в качестве операции массовой отмены, чтобы позволить драйверу отменить все оставшиеся операции ввода/вывода для файла.

Когда все операции ввода/вывода для файла завершены и значение счетчика ссылок на файловый объект принимает нулевое значение, драйвер получает запрос на закрытие. На данном этапе драйвер может безопасно освободить ресурсы, выделенные файлу.

Для драйверов режима ядра существует другая причина, почему Windows поддерживает как очистку, так и закрытие. Запрос на очистку гарантированно прибывает в контексте вызывающего процесса, так что драйвер может выполнить операции очистки в области контекста сеанса. Запросы на закрытие прибывают в полностью произвольном процессе. (*Правин Рао (Praveen Rao), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Запросы на чтение и запись

Приложение обычно издает запросы на чтение, вызывая функцию Windows `ReadFile`, а запросы на запись — вызывая функцию `WriteFile`. Приложение выдает запросы на чтение и запись, чтобы получить или предоставить данные устройству с помощью определенного дескриптора файла. Запросы на чтение содержат буфер, в котором драйвер возвращает запрошенные данные, а запросы на запись содержат буфер с данными для записи. Приложение описывает буфер данных, предоставляя указатель на буфер и размер буфера в байтах.

Менеджер ввода/вывода создает пакет IRP, который описывает запрос, и, в зависимости от типа передачи ввода/вывода для данного стека устройств, организует буфера для буферизованного или прямого ввода/вывода. После этого драйверы устройства обрабатывают запрос.

Так как некоторые устройства могут выполнить запрос на чтение или запись, передавая меньшее количество байтов, чем размер буфера приложения, система возвращает приложению значение количества байтов, переданных в успешной операции чтения или записи.

## Запросы управления вводом/выводом устройства

Приложения издают запросы управления вводом/выводом устройства с помощью функции Windows `DeviceIoControl`. Такие запросы также называются *запросами управления устройством* или *запросами IOCTL*.

Компоненты режима ядра также могут определять и использовать внутренние запросы IOCTL, которые иногда называются закрытыми запросами IOCTL (private IOCTL). Приложения пользовательского режима не могут выдавать внутренние запросы IOCTL.

Запросы IOCTL описывают операции, которые трудно представить в терминах запросов ввода/вывода. Например, драйвер устройства CD-ROM может предоставить способ для открытия и закрытия лотка привода. Подобным образом образцы драйверов Fx2\_Driver и Osrusbf2 поддерживают запрос, посредством которого приложение может контролировать светодиодную линейку на обучающем устройстве USB Fx2.

Когда приложение выдает запрос IOCTL, оно предоставляет код управления, определяющий операцию, которую нужно выполнить. В Windows определены многочисленные стандартные коды управления, которые обычно специфичны для определенного класса устройств. Например, Windows определяет стандартные коды для управления приводов CD и DVD, включая коды для открытия и закрытия лотка привода. Драйверы также определяют специальные коды управления, с помощью которых можно управлять нестандартными устройствами или управлять нестандартными аспектами поведения стандартного устройства. Например:

- ◆ образцы драйверов Fx2\_Driver и Osrusbf2 предназначены для управления нестандартным устройством. В них определены специальные коды, чтобы позволить приложению управлять различными аспектами поведения устройства;
- ◆ примером драйвера, управляющего нестандартными аспектами стандартного устройства, может служить драйвер для привода CD-ROM, предоставляющий специальные коды для управления специфичным для данного привода набором светодиодов.

Кроме кодов управления, существуют другие важные различия между запросами IOCTL и запросами на чтение и запись. А именно:

- ◆ тип ввода/вывода встроен в код управления и для него не требуется быть одинаковым для всех запросов IOCTL, поддерживаемых драйвером;
- ◆ все запросы на чтение и запись должны быть одного типа;
- ◆ для каждого запроса приложение может указывать как входной, так и выходной буфер.

Хотя запросы IOCTL всегда имеют параметры, как для буферов ввода, так и для буферов вывода, не для всех кодов управления требуется указывать оба буфера. Более того, для некоторых кодов запросов IOCTL не требуется указывать буфера совсем. Примером таких кодов может служить код для закрытия лотка привода CD-ROM или DVD;

- ◆ при определенных обстоятельствах буфер вывода запроса IOCTL можно использовать в качестве дополнительного ввода.

### **Проверка достоверности кодов запросов IOCTL**

В начале моей карьеры разработчика драйверов, когда я работал над драйверами для приводов жестких дисков и CD-ROM, я модифицировал фрагмент кода в драйвере класса хранения, чтобы выполнять обработку запросов IOCTL более организованным образом. Некоторые запросы IOCTL существовали в двойных версиях, и одна и та же команда была определена в виде запроса IOCTL как для приводов жестких дисков, так и для приводов CD-ROM. Код управления содержит тип устройства, поэтому числовые значения для приводов жестких дисков и приводов CD-ROM были разными, что усложняло работу с ними.

Поэтому я разбил код управления на его составные компоненты (метод, функция, доступ и устройство), после чего выбросил часть кода для устройства и выполнял проверку только на основе кода для функции.

Но в результате этого подхода появилась ошибка безопасности. Злоумышленник мог открыть устройство, не требуя доступа, после чего выдать запрос IOCTL хранения с обнуленными флагами доступа в коде запроса. Эта команда была бы пропущена менеджером ввода/вывода, т. к. в коде управления указывается, что для нее не требуются никаких привилегий, но мой драйвер все равно бы обработал ее.

Я довольно тем, что мы обнаружили эту ошибку до того, как система была выпущена в производство. Такого рода ошибки делаются только один раз. (Петер Виленд (*Peter Wieland*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Обзор типов запросов ввода/вывода

Каждый тип запроса ввода/вывода сопоставлен коду главной функции пакета IRP. Когда система получает запрос от приложения, менеджер ввода/вывода упаковывает этот запрос в пакет IRP, который он отсылает драйверам устройства. Обработка запросов ввода/вывода от приложений всегда начинается с верха стека устройств, чтобы дать возможность обработать их всем драйверам в стеке.

Кроме запросов ввода/вывода, менеджер ввода/вывода Windows использует пакеты IRP для передачи других типов запросов. WDF поддерживает наиболее употребляемые типы пакетов IRP. Типы пакетов IRP, поддерживаемые WDF, перечислены в табл. 8.1.

**Таблица 8.1. Типы пакетов IRP, поддерживаемые WDF**

Код основной функции пакета IRP WDM	Описание
IRP_MJ_CLEANUP	Поддерживается посредством немедленных обратных вызовов на объекте файла; не ставится в очередь
IRP_MJ_CLOSE	Поддерживается посредством немедленных обратных вызовов на объекте файла; не ставится в очередь
IRP_MJ_CREATE	Поддерживается посредством очередей, как для KMDF, так и для UMDF, и посредством немедленных обратных вызовов на объекте устройства только для KMDF
IRP_MJ_DEVICE_CONTROL	Поддерживается посредством очередей
IRP_MJ_INTERNAL_DEVICE_CONTROL	Поддерживается посредством очередей только для KMDF
IRP_MJ_PNP	Поддерживается посредством специфичных для состояния обратных вызовов на объекте устройства
IRP_MJ_POWER	Поддерживается посредством специфичных для состояния обратных вызовов на объекте устройства
IRP_MJ_READ	Поддерживается посредством очередей
IRP_MJ_SHUTDOWN	Поддерживается для объектов устройств управления (не Plug and Play) только для KMDF
IRP_MJ_SYSTEM_CONTROL	Поддерживается посредством объектов интерфейса WMI только в KMDF
IRP_MJ_WRITE	Поддерживается посредством очередей

Обратные вызовы объекта устройства, специфичные для состояния, описываются в *главе 7*. Каким образом драйверы KMDF могут обрабатывать дополнительные типы пакетов IRP, выходя за пределы инфраструктуры, описывается в *главе 14*.

## Типы передач ввода/вывода

Windows поддерживает следующие три механизма передачи данных, также называемых типами передач ввода/вывода (I/O transfer types):

- ◆ при буферизованном вводе/выводе обрабатывается копия данных пользователя;
- ◆ при прямом вводе/выводе выполняется прямой доступ к пользовательским данным с помощью списков MDL (Memory Descriptor List, список дескрипторов памяти) и указателей режима ядра;

- ◆ при вводе/выводе типа METHOD\_NEITHER (ни буферизованный, ни прямой) доступ к данным пользователя осуществляется посредством указателей пользовательского режима.

Драйверы WDF поддерживают все три типа ввода/вывода. Но необходимо избегать применения типа ввода/вывода METHOD\_NEITHER по причине свойственных ему сложностей с проверкой достоверности данных и использования указателей пользовательского режима.

Для запросов IOCTL тип передачи включен в сам код запроса, поэтому запросы IOCTL могут быть любого из трех типов и использование одного и того же типа передач ввода/вывода для всех запросов IOCTL не является обязательным. Все запросы на чтение и запись должны применять один и тот же тип передачи ввода/вывода, т. к. тип ввода/вывода для запросов на чтение и запись ассоциирован с самим объектом устройства.

Драйвер KMDF должен указывать тип передачи ввода/вывода, поддерживаемый каждым объектом устройства для запросов на чтение и запись. Чтобы установить тип ввода/вывода, драйвер KMDF вызывает метод `WdfDeviceInitSetIoType` в функции обратного вызова `EvtDriverDeviceAdd` перед созданием объекта устройства. Драйвер может указать одну из следующих констант перечисления `WDF_DEVICE_IO_TYPE`:

- ◆ `WdfDeviceIoBuffered`;
- ◆ `WdfDeviceIoDirect`;
- ◆ `WdfDeviceIoNeither`.

По умолчанию применяется константа `WdfDeviceIoBuffered`.

Драйвер Osrusbf2 устанавливает буферизованный тип ввода/вывода, как показано в следующем операторе из файла исходного кода Device.c:

```
WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoBuffered);
```

Установить тип ввода/вывода с помощью метода `WdfDeviceInitSetIoType` в драйвере фильтра не получится. Для драйверов фильтра инфраструктура применяет тип ввода/вывода, указываемый следующим нижележащим драйвером в стеке устройств.

### **Почему тип ввода/вывода не играет роли в UMDF?**

В KMDF драйверы могут избрать прямой или буферизованный тип ввода/вывода для запросов на чтение и запись. Почему же такой опции нет для драйверов UMDF?

Причиной этому является тот факт, что все операции ввода/вывода в UMDF буферизованного типа.

Отражатель копирует данные запроса из указанного вызывающим клиентом буфера или списка MDL в буфер в хост-процессе; то же самое происходит и в обратном направлении. (К счастью, современные процессоры выполняют задачу копирования данных очень эффективно.) Поэтому для драйвера нет особой надобности указывать требуемый тип ввода/вывода. Так как отражатель является самым высшим драйвером в стеке устройств режима ядра, для запросов IOCTL он может применять тип ввода/вывода METHOD\_NEITHER. Отражатель аккуратно копирует данные прямым образом между буферами приложения и хост-процессом.

Я предполагаю, что вы хотите сказать: "Ну, для чтения и записи это подойдет, но вы ведь только что сказали мне, что запросы IOCTL устанавливают свой тип передачи в коде управления. Разве это не приводит к выполнению лишних операций копирования?" В Windows XP для буферизованных и прямых запросов IOCTL выполняется лишнее копирование. А в Windows Vista и последующих выпусках отражатель указывает менеджеру ввода/вывода рассматривать все получаемые им запросы IOCTL как запросы типа METHOD\_NEITHER, независимо от типа передачи. Таким образом, мы можем избежать лишнего копирования в новых выпусках этой операционной системы. (*Питер Виланд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Буферизованный ввод/вывод

Когда менеджер ввода/вывода выдает запрос на буферизованный ввод/вывод, пакет IRP содержит внутреннюю копию буфера вызывающего клиента, а не сам буфер клиента. Для запросов на запись менеджер ввода/вывода копирует данные из буфера клиента во внутренний буфер, а для запросов на чтения — из внутреннего буфера в буфер клиента.

Драйвер WDF получает объект запроса WDF, который, в свою очередь, содержит встроенный объект памяти WDF. Объект памяти содержит адрес буфера, с которым должен работать драйвер.

## Прямой ввод/вывод

Когда менеджер ввода/вывода выдает запрос прямого ввода/вывода, пакет IRP содержит адрес списка MDL, который описывает буфер запроса.

Для драйверов UMDF отражатель проверяет размер буфера и режим доступа и копирует эту информацию в буфер в хост-процессе. Драйвер получает этот новый буфер в объекте запроса WDF. Драйверы UMDF могут выполнять чтение и запись в этот буфер, точно так же, как и в любой другой буфер.

Для запросов на чтение и запись отражатель копирует данные между буфером клиента и хост-процессом.

Для запросов IOCTL отражатель поступает следующим образом.

- ◆ Если в коде управления указывается метод METHOD\_OUT\_DIRECT, то отражатель копирует содержимое внутреннего буфера в хост-процесс.
- ◆ Если в коде управления указывается метод METHOD\_IN\_DIRECT, то отражатель копирует как входной, так и выходной буферы в хост-процесс, т. к. выходной буфер может также использоваться в качестве дополнительного буфера ввода. По завершению запроса METHOD\_IN\_DIRECT отражатель копирует содержимое выходного буфера из хост-процесса обратно в первоначальный пакет IRP.

Для драйверов KMDF объект памяти, встроенный в запрос WDF, содержит адрес списка MDL, который описывает буфер запроса. В списке MDL указывается размер буфера и его виртуальный адрес, а также физические страницы в буфере. Прежде чем выдать пакет IRP, менеджер ввода/вывода блокирует эти физические страницы и снимает блокировку при завершении выполнения пакета IRP. Драйверы KMDF могут применять методы WDF для чтения и записи в буфер, или же могут выполнять чтение и запись буфера непосредственно, используя список MDL. KMDF обеспечивает получение драйвером указателя на виртуальный системный адрес для списка MDL, так что драйверу не нужно соотносить адрес, как это приходится делать драйверам WDM.

## Ввод/вывод типа METHOD\_NEITHER

Когда менеджер ввода/вывода выдает запрос IOCTL, указывающий тип передачи METHOD\_NEITHER, пакет IRP содержит указатель на буфер пользовательского режима, предоставленный приложением, издавшим запрос.

### Тип ввода/вывода METHOD\_NEITHER в UMDF

UMDF предоставляет частичную поддержку ввода/вывода типа METHOD\_NEITHER посредством директивы `UmdfMethodNeitherAction` в INF-файле. Эта директива устанавливается в реестре

значение, которое считывается отражателем, чтобы определить, каким образом обрабатывать запросы типа `METHOD_NEITHER`. По умолчанию отражатель завершает неудачей все запросы типа `METHOD_NEITHER`.

При разрешенном вводе/выводе типа `METHOD_NEITHER` отражатель копируют данные и размеры буферов из пакета IRP в хост-процесс. Но отражатель может успешно скопировать эту информацию только для действительных указанных пользователем адресов и размеров буферов. В некоторых запросах ввода/вывода параметр размера буфера в действительности не указывает размер буфера, а предоставляет вместо этого какие-либо иные специфичные для кода управления данные. Если конструкция кода IOCTL такова, что отражатель не может определить, правильно ли сформирован запрос, нельзя полагать, что данные и размер буферов, полученные драйвером UMDF, будут правильные.

Кроме этого, запросы IOCTL типа `METHOD_NEITHER` представляют собой угрозу безопасности. Менеджер ввода/вывода не может проверить достоверность размера входного и выходного буферов, таким образом оставляя драйвер уязвимым к атаке.

В общем случае, тип ввода/вывода `METHOD_NEITHER` следует разрешать только в тех случаях, когда драйвер требует его, и даже если запросы IOCTL, поддерживаемые драйвером, используют параметры предназначенным образом.

Ввод/вывод типа `METHOD_NEITHER` для драйвера пользовательского режима можно разрешить, включив следующую директиву в INF-файл драйвера:

```
UmdfMethodNeitherAction = Действие
```

Здесь `Действие` может быть `Copy` или `Reject`.

- ◆ Действие `Copy` указывает, что объект запроса WDF содержит копию данных в буфере пользователя.
- ◆ Действие `Reject` означает, что отражатель должен завершать запрос неудачей.

Директива `UmdfMethodNeitherAction` не является обязательной. Действие по умолчанию — `Reject`.

## Тип ввода/вывода `METHOD_NEITHER` в KMDF

Драйвер KMDF может получать запросы ввода/вывода типа `METHOD_NEITHER` от клиентов пользовательского режима или режима ядра.

Если запрос исходит от клиента пользовательского режима, менеджер ввода/вывода передает указатель на буфер пользовательского режима. Прежде чем обращаться к объекту памяти WDF, драйвер должен проверить достоверность адреса, блокировать страницы памяти буфера и получить любые данные, требуемые драйвером для управления операцией (например, размер буферов и указатели буферов). Драйвер получает данные путем копирования их в безопасное место в режиме ядра, где они не могут быть модифицированы клиентом. Таким образом, обеспечивается, что данные в своей работе драйвер будет использовать действительные. Драйверы KMDF должны выполнять проверку данных в обратном вызове функции `EvtIoInCallerContext`. Каким образом реализовать этот обратный вызов, объясняется в разд. "Извлечение буферов в драйверах KMDF" далее в этой главе.

К запросам, исходящим от клиента режима ядра, эти требования не распространяются. Драйвер может обращаться к встроенному объекту памяти WDF напрямую.

## Поток запросов ввода/вывода

На рис. 8.1 показан общий путь, по которому следуют запросы ввода/вывода от приложения через систему и через стек устройств. Стек устройств может содержать любую комбинацию драйверов UMDF, KMDF и других драйверов режима ядра. Подробности обработки ввода/вывода в драйверах UMDF и KMDF описываются далее в этой главе.

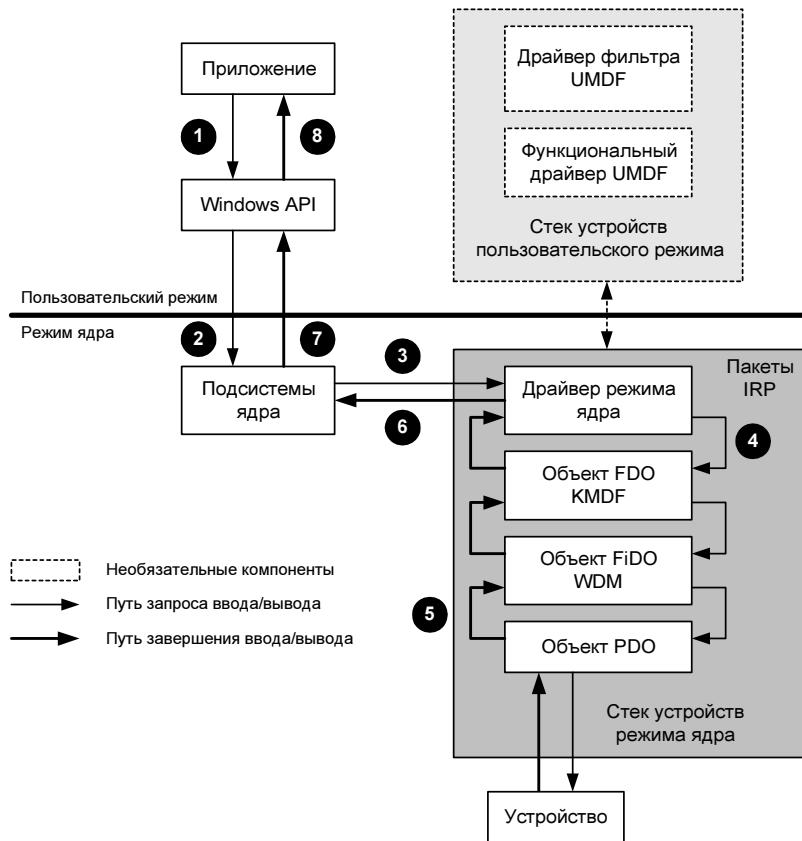


Рис. 8.1. Путь запросов ввода/вывода от приложения к устройству

На рис. 8.1 цифрами обозначены следующие основные участки пути:

1. Приложение выдает запрос ввода/вывода, вызывая функцию Windows API.
2. Windows API вызывает подсистемы ядра, которые вызывают соответствующую функцию обработки диспетчера ввода/вывода.
3. Менеджер ввода/вывода создает пакет IRP, описывающий запрос ввода/вывода, и посыпает его верхнему драйверу в стеке устройств режима ядра целевого устройства.

Верхний драйвер в стеке устройств режима ядра обрабатывает и завершает запрос, если он может это.

Если устройство имеет драйвер UMDF, верхним драйвером в стеке устройств режима ядра является отражатель, и пакет IRP получает его объект устройства Up. Отражатель

упаковывает запрос и посыпает его хост-процессу драйвера пользовательского режима, для обработки драйвером UMDF. Если устройство не имеет драйвера UMDF, не выполняется никакой обработки пользовательского режима.

Подробности обработки пользовательского режима описываются в разд. "Путь запроса ввода/вывода через стек устройства UMDF" далее в этой главе.

Если в результате обработки либо стеком устройств UMDF, либо драйвером режима ядра пакет IRP завершен, шаги 4 и 5 пропускаются и обработка продолжается с шага 6.

4. Если пакет IRP еще не завершен, драйвер режима ядра отправляет его следующему нижележащему драйверу режима ядра в стеке устройств. Этот драйвер обрабатывает пакет IRP и, в свою очередь, отправляет его следующему нижележащему драйверу в стеке устройств, и т. д. до тех пор, пока драйвер не завершит пакет IRP.

Любой драйвер, который посыпает пакет IRP вниз по стеку, может зарегистрировать обратный вызов завершения ввода/вывода и иметь возможность снова обрабатывать пакет, после завершения обработки нижними драйверами.

Наконец, пакет IRP достигает драйвера, который завершает его обработку. На рис. 8.1 это драйвер объекта PDO, но это мог быть любой другой драйвер в стеке.

5. После завершения обработки пакета IRP менеджер ввода/вывода вызывает функции обратного вызова, установленные драйверами при прохождении запроса вниз по стеку, в порядке, обратном порядку их установки.

Драйвер, который завершает запрос IRP, не требует обратного вызова завершения ввода/вывода, т. к. он уже "знает", что запрос завершен и завершил свою часть его обработки.

6. Менеджер ввода/вывода извлекает из пакета IRP статус ввода/вывода, преобразовывает его в код ошибки Windows, возвращает информацию о завершении и, если необходимо, копирует данные в буфер данных запрашивающего приложения.
7. После этого менеджер ввода/вывода возвращает управление Windows API.
8. Windows API возвращает результаты обработки запрашивающему приложению.

## Путь запроса ввода/вывода через стек устройств UMDF

На рис. 8.2 показано протекание запроса ввода/вывода от отражателя через стек устройств UMDF.

На рис. 8.2 цифрами обозначены следующие основные участки пути протекания запроса:

1. Менеджер ввода/вывода доставляет пакет IRP верхнему драйверу в стеке устройств режима ядра. Если устройство имеет один или несколько драйверов UMDF, верхним драйвером в стеке устройств режима ядра является отражатель, и получателем запросов ввода/вывода, отправленных менеджером ввода/вывода, является его объект устройства Up.
2. Отражатель упаковывает запрос и посыпает его хост-процессу драйвера, который создает пакет IRP пользовательского режима. В хост-процессе драйвера инфраструктура проверяет достоверность параметров запроса и обрабатывает запрос, начиная с верхнего драйвера в стеке пользовательского режима. На рисунке показаны два драйвера UMDF, но устройство может иметь только один или несколько таких драйверов.

Если, как показано на рисунке, драйвер UMDF является драйвером фильтра, который не обрабатывает запросы данного типа, инфраструктура пересыпает пакет IRP пользователь-

тельского режима стандартному получателю ввода/вывода, у которого драйвер является следующим нижележащим драйвером в стеке устройств. Инфраструктура потом определяет, может ли драйвер обрабатывать такие запросы. Каким образом инфраструктура делает это определение, описывается в разд. "Поток запросов ввода/вывода в инфраструктурных стеках" далее в этой главе.

Если драйвер UMDF является функциональным драйвером, который не обрабатывает запросы данного типа, инфраструктура не выполняет запрос, возвращая статус `STATUS_INVALID_DEVICE_REQUEST`. Шаги 3—6 пропускаются, и обработка продолжается с шага 7.

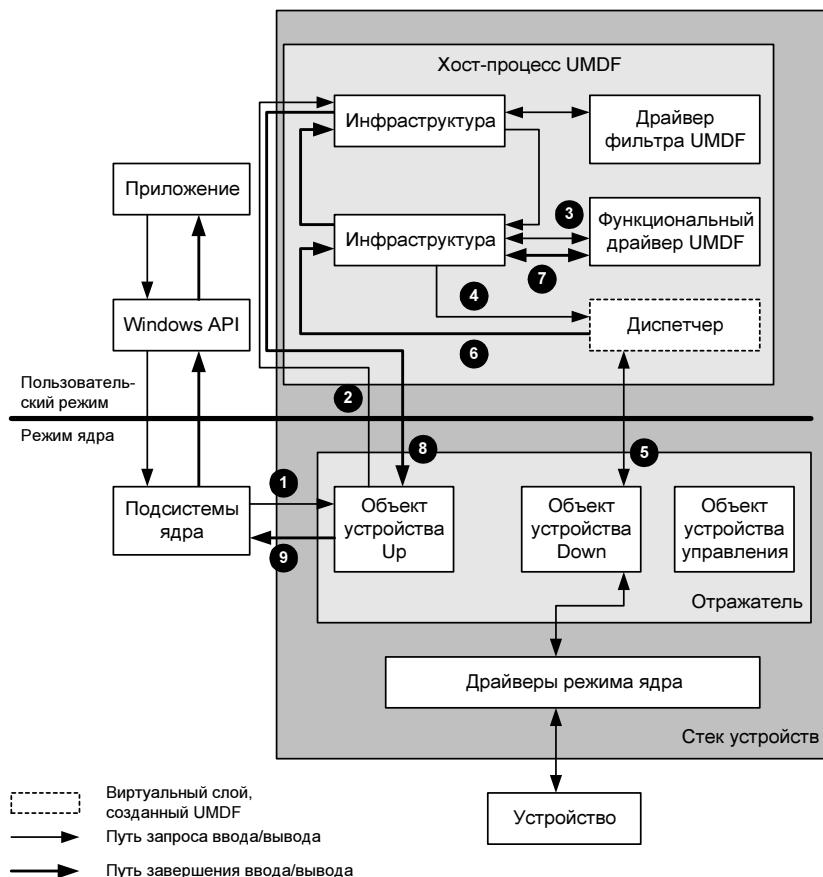


Рис. 8.2. Путь запроса ввода/вывода через стек устройств UMDF

- Когда запрос доходит до драйвера, который может обработать его, инфраструктура создает объект запроса WDF и добавляет его в соответствующую очередь или вызывает соответствующий метод обратного вызова. Драйвер потом обрабатывает запрос.

Если драйвер не может завершить запрос WDF, он посыпает его стандартному получателю ввода/вывода, которым является следующий нижележащий драйвер в стеке. Это действие передает запрос обратно инфраструктуре, которая сохраняет указатель на объект запроса WDF, чтобы она могла определить позже, зарегистрировал ли драйвер обратный

вызов завершения запроса. Инфраструктура вызывает для обработки запроса следующий драйвер (если таковой имеется) в стеке устройств пользовательского режима, и т. д. вниз по стеку, таким же образом, как и в драйверах режима ядра.

Любой драйвер UMDF, который посыпает запрос стандартному получателю ввода/вывода, может зарегистрировать обратный вызов завершения ввода/вывода, чтобы получить извещение, когда пакет завершен.

Если драйвер может завершить запрос WDF, он это делает, вызывая метод семейства `IWDFIoRequest::CompleteXxx` на объекте запроса. Когда драйвер завершает запрос, обработка продолжается с шага 7.

4. Если все драйверы UMDF обрабатывают запрос и потом посыпают его стандартному получателю ввода/вывода, инфраструктура вызывает менеджер ввода/вывода. Менеджер ввода/вывода выдает новый запрос ввода/вывода Windows для объекта устройства Down, так что этот объект получает другой пакет IRP, а не первоначальный пакет IRP, посланный приложением. Объект устройства Down является стандартным получателем ввода/вывода для нижнего драйвера в стеке устройств пользовательского режима.
5. Объект устройства Down получает новый пакет IRP от менеджера ввода/вывода и посыпает его следующему нижележащему драйверу режима ядра, который расположен сразу же под отражателем. После этого обработка продолжается сквозь стек устройств режима ядра, как описано в предыдущем разделе.
6. Когда новый пакет IRP завершен, менеджер ввода/вывода получает информацию о статусе и завершении таким же образом, как и о любом другом приложении пользовательского режима. Он извещает инфраструктуру, чтобы та могла управлять обработкой завершения запроса WDF и соответствующего пакета IRP пользовательского режима сквозь стек устройств UMDF.
7. Инфраструктура вызывает функции обратных вызовов завершения ввода/вывода (если таковые имеются), которые были зарегистрированы драйверами UMDF при прохождении пакета IRP пользовательского режима вниз по стеку. Эти обратные вызовы выполняются в порядке, обратном тому, в котором они были установлены.
8. После возвращения управления последним обратным вызовом завершения UMDF, инфраструктура передает отражателю статус завершения и данные.
9. Отражатель завершает первоначальный пакет IRP и передает его менеджеру ввода/вывода, который продолжает процесс завершения обратно к приложению, как описано в предыдущем разделе.

## Путь запросов ввода/вывода сквозь драйвер KMDF

На рис. 8.3 показан путь протекания запросов сквозь драйвер KMDF. В данном примере стек устройств содержит функциональный драйвер KMDF, расположенный под драйвером фильтра. Но положение драйвера в стеке не влияет на процесс обработки, которая выполняется таким же образом.

На рис. 8.3 цифрами обозначены следующие основные шаги обработки:

1. Верхний драйвер в стеке устройств режима ядра получает пакет IRP от менеджера ввода/вывода и обрабатывает его, как было описано ранее. На рис. 8.3 верхним драйвером является драйвер фильтра, который передает запрос вниз по стеку следующему нижележащему драйверу, которым является функциональный драйвер KMDF.

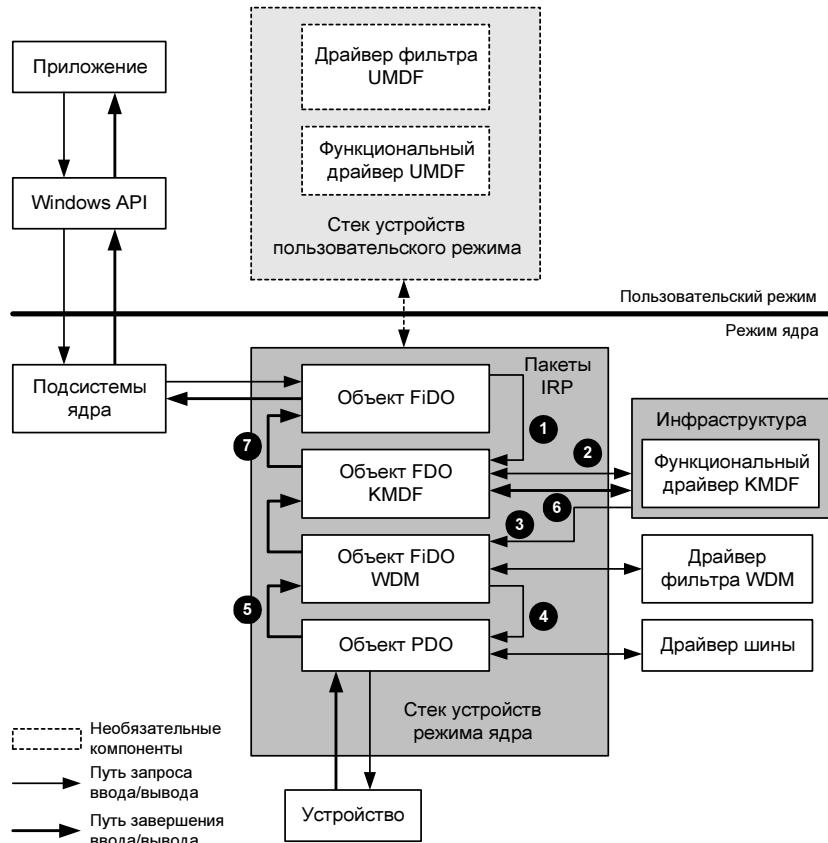


Рис. 8.3. Путь запросов ввода/вывода сквозь драйвер KMDF

- В стеке устройств функциональный драйвер KMDF представлен объектом FDO, созданным инфраструктурой. Инфраструктура перехватывает пакет IRP, анализирует код основной функции и определяет, может ли этот драйвер обработать данный пакет IRP. Каким образом инфраструктура делает это определение, описывается в разд. "Поток запросов ввода/вывода в инфраструктурах" далее в этой главе.

Для функционального драйвера KMDF или драйвера шины KMDF инфраструктура завершает запрос неудачей, возвращая статус `STATUS_INVALID_DEVICE_REQUEST`, после чего обработка продолжается с шага 6.

В случае драйвера фильтра KMDF, который не обрабатывает запросы данного типа, инфраструктура отправляет запрос следующему нижележащему драйверу и обработка продолжается с шага 4.

В противном случае инфраструктура создает объект запроса WDF и выполняет обратный вызов функции драйвера KMDF или добавляет запрос в очередь, в зависимости от того, которое из этих действий является уместным в данном случае.

- Драйвер KMDF обрабатывает запрос. Если драйвер завершает запрос, он вызывает метод `WdfRequestCompleteXxx`. После этого инфраструктура начинает обработку завершения с шага 7; шаги 4—6 пропускаются.

Если драйвер не может завершить запрос, он посыпает его стандартному получателю ввода/вывода, которым является следующий нижележащий драйвер в стеке. Если после выполнения завершения нижележащими драйверами, драйверу требуется выполнить дополнительную обработку, то он регистрирует в KMDF обратный вызов функции завершения. В свою очередь, KMDF регистрирует процедуру завершения в пакете IRP.

4. Запрос обрабатывается следующим нижележащим драйвером, и т. д. вниз по стеку.
5. После завершения запроса менеджер ввода/вывода делает обратные вызовы функций завершения, зарегистрированных в пакете IRP. Если драйвер KMDF зарегистрировал обратный вызов функции завершения, инфраструктура получает обратно пакет IRP, когда менеджер ввода/вывода вызывает инфраструктурную процедуру завершения.
6. Инфраструктура делает обратный вызов функции завершения для объекта запроса WDF, чтобы драйвер мог выполнить постобработку запроса.
7. Инфраструктура обеспечивает, чтобы первоначальный пакет IRP содержал информацию о статусе и завершении и все запрошенные данные. После этого она удаляет объект запроса WDF и возвращает управление менеджеру ввода/вывода, который продолжает процесс выполнения завершения запроса вверх по стеку, в конце концов, доходя до запрашивающего приложения.

## Обработка завершения запросов ввода/вывода

По завершению запроса ввода/вывода, как WDF, так и Windows выполняют обработку завершения. Инфраструктуры выполняют очистку объекта запроса WDF и возвращают информацию менеджеру ввода/вывода. В свою очередь, менеджер ввода/вывода копирует запрошенные данные в буфер пользователя и возвращает информацию завершения.

### Обработка завершения запросов ввода/вывода UMDF

Когда драйвер UMDF завершает запрос ввода/вывода WDF, инфраструктура действует следующим образом:

1. Копирует возвращенные данные, статус ввода/вывода и информацию завершения, как требуется объектом запроса WDF, в пакет IRP пользовательского режима.
2. Выполняет обратный вызов функции завершения для объекта запроса WDF, если драйвер зарегистрировал такой обратный вызов. Находящиеся ниже буферы и пакет IRP WDM продолжают оставаться действительными.
3. Уничтожает сам объект запроса WDF, освобождая хранилище контекста, которое было сопоставлено с объектом.
4. Извещает отражатель, который копирует возвращенные данные в нижний пакет IRP WDM и завершает пакет IRP WDM.

### Обработка завершения запросов ввода/вывода KMDF

Когда драйвер KMDF завершает запрос ввода/вывода WDF, инфраструктура действует следующим образом:

1. Заполняет поля статуса ввода/вывода и информации о завершении в пакете IRP данными, взятыми с соответствующих полей в объекте запроса WDF.
2. Выполняет обратный вызов функции завершения для объекта запроса WDF, если драйвер зарегистрировал такой обратный вызов. Находящийся ниже в стеке пакет IRP WDM продолжает оставаться действительным.

3. Завершает пакет IRP, ассоциированный с запросом WDF, после возвращения управления функцией обратного вызова.
4. Освобождает свою ссылку на объект запроса WDF. Когда значение счетчика ссылок становится нулевым, инфраструктура выполняет обратный вызов функции уничтожения, если драйвер зарегистрировал такой вызов.
5. Уничтожает сам объект запроса WDF, освобождая хранилище контекста, которое было сопоставлено с объектом.

Одним словом, инфраструктура управляет очисткой и уничтожением объекта запроса WDF таким же образом, как и для любого другого объекта WDF.

## **Обработка завершения запросов ввода/вывода в Windows**

По завершению пакета IRP Windows возвращает до трех элементов данных в поток, издавший запрос.

### ◆ Статус ввода/вывода.

Значение статуса ввода/вывода может быть следующим:

- `HRESULT` для драйверов UMDF;
- `NTSTATUS` для драйверов KMDF.

Менеджер ввода/вывода преобразовывает результат и возвращает код ошибки Windows приложению, издавшему запрос.

### ◆ Информация завершения.

Это значение указывает количество байтов, переданных в результате успешного запроса на чтение, запись или IOCTL.

Данная информация возвращается приложению Windows в параметре `lpNumberOfBytesRead` при вызове функции `ReadFile`, в параметре `lpNumberOfBytesWritten` при вызове функции `WriteFile` или в параметре `lpBytesReturned` при вызове функции `DeviceIoControl`.

### ◆ Запрошенные данные.

Для запросов на чтение и определенных запросов IOCTL система возвращает данные в буфере, предоставленном вызывающим приложением.

## **Поток запросов ввода/вывода в инфраструктурах**

В инфраструктуре внутренний маршрутизатор пакетов IRP анализирует код основной функции в каждом входящем пакете IRP, чтобы определить, какой из следующих внутренних компонентов должен обрабатывать пакет IRP.

### ◆ Обработчик запросов ввода/вывода.

Обработчик запросов ввода/вывода направляет запросы ввода/вывода драйверу, управляет отменой и завершением запросов ввода/вывода, а также работает совместно с обработчиком Plug and Play и энергопотребления, чтобы обеспечить состояние устройства совместимое с вводом/выводом, выполняемым устройством.

### ◆ Обработчик Plug and Play и энергопотребления.

Обработчик Plug and Play и энергопотребления управляет процессом Plug and Play и управлением энергопотребления для драйвера, применяя для этого свои внутренние ма-

шины состояний и реализованные в драйверах обратные вызовы функций. Обработчик может внутренне обрабатывать и завершать прибывающие запросы или пересыпать их другим драйверам в системе для дополнительной обработки и завершения в итоге обработке.

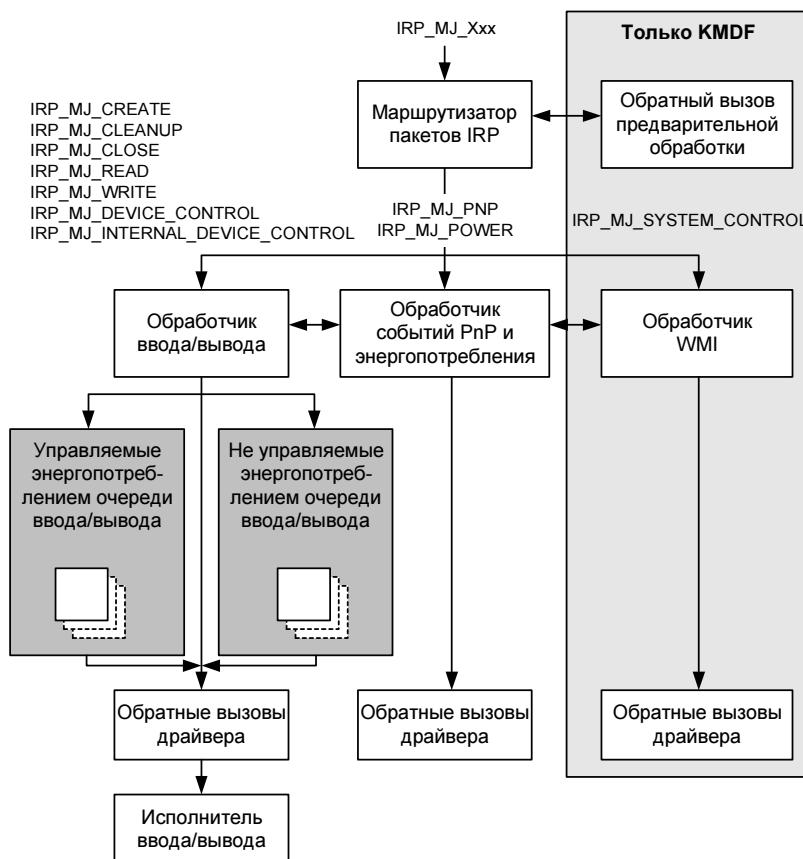
Дополнительная информация о машинах состояний Plug and Play и энергопотреблении предоставления в [главе 7](#).

- ◆ Обработчик WMI (применимо только к KMDF).

Обработчик WMI поддерживает все типы запросов WMI и предоставляет стандартную реализацию интерфейса WMI, так что драйверы KMDF, не предоставляющие данных WMI, не должны регистрироваться в качестве поставщиков WMI.

Поддержка драйверами KMDF запросов WMI описывается в главе 12.

Обе инфраструктуры обрабатывают запросы подобным образом на конвейере запросов. Блок-схема конвейера обработки запросов WDF показана на рис. 8.4. В сером прямоугольнике справа показаны возможности, поддерживаемые только в KMDF. Остальная часть диаграммы применима как к UMDF, так и к KMDF.



**Рис. 8.4.** Конвейер обработки запросов WDF

Когда пакет IRP поступает на конвейер, маршрутизатор пакетов IRP анализирует код основной функции, чтобы определить, по какому маршруту направить запрос на обработку.

## Предварительная обработка пакетов IRP

Драйверы KMDF могут зарегистрировать обратный вызов функции для предварительной обработки пакетов IRP, не обрабатываемых KMDF. Если драйвер KMDF зарегистрировал такой вызов для кода основной функции данного пакета IRP, маршрутизатор пакетов IRP активирует этот обратный вызов. После завершения предварительной обработки, пакет IRP обычно возвращается маршрутизатору. Обработка драйвером KMDF пакетов IRP, не обрабатываемых инфраструктурой KMDF, описывается в главе 14.

Драйверы UMDF не могут обрабатывать пакеты IRP, не обрабатываемые инфраструктурой. Поэтому в драйверах UMDF нет предварительной обработки.

## Маршрутизация пакетов IRP к внутреннему обработчику запросов

Как для UMDF, так и для KMDF маршрутизатор пакетов IRP посыпает пакеты IRP одному из внутренних обработчиков инфраструктуры:

- ◆ пакеты IRP с кодами основной функции `IRP_MJ_PNP` и `IRP_MJ_POWER` направляются обработчику Plug and Play и энергопотребления;
- ◆ пакеты IRP со следующими кодами основной функции направляются обработчику ввода/вывода:
  - `IRP_MJ_CREATE`;
  - `IRP_MJ_WRITE`;
  - `IRP_MJ_CLEANUP`;
  - `IRP_MJ_DEVICE_CONTROL`;
  - `IRP_MJ_CLOSE`;
  - `IRP_MJ_INTERNAL_DEVICE_CONTROL` (только KMDF);
  - `IRP_MJ_READ`;
- ◆ пакеты IRP с кодом основной функции `IRP_MJ_SYSTEM_CONTROL` отсылаются обработчику WMI (только для KMDF);
- ◆ пакеты IRP с другими кодами основной функции отсылаются стандартному обработчику, который не показан на рис. 8.4.

Если объект устройства представляет драйвер фильтра, стандартный обработчик пересыпает такие пакеты IRP следующему нижележащему драйверу. В противном случае он завершает пакет IRP, возвращая статус `STATUS_INVALID_DEVICE_REQUEST`.

## Ход операций в обработчике запросов ввода/вывода

Когда обработчик запросов ввода/вывода получает новый запрос, он сначала проверяет, поддерживает ли WDF этот тип запроса и обрабатывается ли этот тип запроса драйвером. Для драйверов Plug and Play инфраструктура WDF поддерживает следующие типы запросов ввода/вывода (табл. 8.2).

**Таблица 8.2.** Поддерживаемые WDF типы запросов ввода/вывода

Тип запроса ввода/вывода	Код основной функции пакета IRP
На создание	IRP_MJ_CREATE
На очистку	IRP_MJ_CLEANUP
На закрытие	IRP_MJ_CLOSE
На чтение	IRP_MJ_READ
На запись	IRP_MJ_WRITE
IOCTL	IRP_MJ_DEVICE_CONTROL
Внутренний IOCTL	IRP_MJ_INTERNAL_DEVICE_CONTROL (только для KMDF)

Если инфраструктура не поддерживает, а драйвер не обрабатывает данный тип запросов ввода/вывода, обработчик ввода/вывода завершает запрос, возвращая статус STATUS\_INVALID\_DEVICE\_REQUEST. После этого пакет IRP возвращается менеджеру ввода/вывода, который возвращает статус неуспешного завершения запроса приложению, издавшему запрос.

### Проверка достоверности параметров

Если WDF поддерживает и драйвер обрабатывает данный тип запросов ввода/вывода, следующим этапом обработчик ввода/вывода проверяет параметры в пакете IRP на соответствие типу запроса.

- ◆ **Запросы на чтение и запись.** Обработчик ввода/вывода проверяет, не содержит ли пакет IRP буфер данных размером в ноль байтов. Хотя менеджер ввода/вывода считает буфера размером в ноль байтов допустимыми, драйверы редко обрабатывают такие запросы, а если и обрабатывают их, то обычно делают это неправильно. Соответственно, по умолчанию обработчик ввода/вывода завершает такие запросы на чтение и запись с успехом, указывая передачу нулевого количества байтов. Но драйвер может сконфигурировать свои очереди для получения запросов с буферами нулевого размера.
- ◆ **Запросы IOCTL.** Обработчик ввода/вывода проверяет буфера ввода и вывода на присутствие запросов IOCTL, требующих ввода/вывода прямого типа (т. е. типов METHOD\_IN\_DIRECT или METHOD\_OUT\_DIRECT). Если один из буферов недействительный, обработчик ввода/вывода завершает запрос, возвращая соответствующий код ошибки и указывая, что было передано нулевое количество байтов.

Хотя инфраструктура проверяет буфера на достоверность, для данного типа ввода/вывода она не может определить отсутствующий буфер или правильный размер буфера, т. к. такие запросы являются специфичными для устройства. Драйвер должен проверять действительность параметров с учетом требований каждого отдельного запроса.

### Обработка запросов в обработчике ввода/вывода

Если пакет IRP успешно проходит эти проверки, обработчик ввода/вывода определяет следующее действие в зависимости от типа пакета IRP, типа драйвера и конфигурации очереди драйвера. Обработчик ввода/вывода выполняет такую последовательность операций до тех пор, пока он не завершит пакет IRP, отоследит его получателю ввода/вывода или создаст и пошлет драйверу запрос WDF, представляющий этот пакет IRP.

- ◆ **Если возможно, обрабатывает запрос вместо драйвера.** Если обработчик ввода/вывода может обработать и завершить пакет IRP вместо драйвера, он так и делает. WDF обычно обрабатывает вместо драйвера запросы на создание, очистку и закрытие, хотя драйвер может предоставить обратные вызовы для обработки таких запросов самому. По умолчанию WDF завершает такие запросы для функциональных драйверов, возвращая статус успешного выполнения, и пересыпает их получателю ввода/вывода для драйверов фильтра. Драйвер может подменить эту стандартную обработку при создании объекта устройства.
- ◆ **Если уместно, посыпает запрос стандартному получателю ввода/вывода.** Если обработчик ввода/вывода не может завершить пакет IRP для драйвера, он определяет, уместно ли послать пакет IRP стандартному получателю ввода/вывода. Для драйвера фильтра обработчик ввода/вывода проверяет наличие сконфигурированной драйвером очереди для обработки запросов данного типа. Для других драйверов обработчик ввода/вывода отсылает пакет IRP стандартному получателю ввода/вывода драйвера, как показано на рис. 8.1 и 8.2 ранее в этой главе.
- ◆ **Направляет запрос драйверу.** Если пакет IRP не был завершен или переслан, обработчик ввода/вывода создает новый объект запроса WDF и направляет его драйверу.

Этот объект запроса WDF описывает и управляет обработкой запроса и пересыпает данные из первоначального пакета IRP драйверу WDF.

После создания объекта запроса WDF обработчик ввода/вывода пересыпает запрос драйверу, выполняя обратный вызов соответствующей функции драйвера или помещая запрос в одну из очередей драйвера. В зависимости от того, как очереди драйвера обрабатывают запросы ввода/вывода, помещение запроса в очередь может активировать вызов одной из функций обратного вызова драйвера для событий ввода/вывода.

## Объекты запросов ввода/вывода

Когда прибывает запрос, который драйвер WDF способен обработать, WDF создает объект запроса ввода/вывода для представления нижележащего в стеке (подлегающего) пакета IRP, помещает объект в очередь и, в конечном счете, отправляет запрос драйверу. Драйвер получает запрос в виде параметра для функции обратного вызова по событию ввода/вывода или вызывая метод очереди.

- ◆ В UMDF объект запроса ввода/вывода предоставляет интерфейс `IWDFIoRequest`.
- ◆ В KMDF запрос ввода/вывода представляется объектом типа `WDFREQUEST`, и драйвер вызывает методы семейства `WdfRequestXxx` для работы с ним.

Драйвер может вызывать методы WDF для получения информации о запросе, такой как, например, тип запроса, параметры, буферы данных и ассоциированный файловый объект.

Как и со всеми прочими объектами WDF, для объекта запроса WDF ведется подсчет ссылок, и он может иметь собственную область контекста объекта. После завершения драйвером запроса WDF, инфраструктура освобождает свою ссылку на объект запроса WDF и на все его дочерние объекты и выполняет обратный вызов их функций очистки, если таковые имеются. После возвращения управления функциями зачистки драйвер больше не имеет доступа к подлегающему в стеке пакету IRP.

После того как драйвер завершил запрос, он не должен пытаться обращаться к объекту запроса или любому из его дочерних объектов, кроме как для освобождения всех еще существующих ссылок.

### Внимание!

Подсчет ссылок ведется только для объекта запроса ввода/вывода WDF, а не для нижележащего пакета запроса IRP WDM. После того как драйвер завершит запрос ввода/вывода, указатель на пакет IRP перестает быть достоверным и драйвер не может обращаться к пакету IRP.

Драйвер может создавать собственные объекты запроса ввода/вывода, чтобы запросить ввод/вывод у другого устройства или для того, чтобы разбить запрос ввода/вывода на несколько меньших запросов перед тем, как завершить его.

Информация о создании объектов запроса ввода/вывода драйвером предоставляется в *главе 9*.

## Буферы ввода/вывода и объекты памяти

Запросы ввода/вывода на чтение, запись и IOCTL содержат буферы для входных и выходных данных, связанных с запросом. Инфраструктура WDF создает объекты памяти, чтобы инкапсулировать эти буфера, и встраивает эти объекты памяти в создаваемые ею объекты запроса ввода/вывода. Каждый объект памяти содержит размер буфера, который он представляет. Методы WDF, которые копируют данные в буфер и из буфера, проверяют размер переданных данных при каждой операции передачи, с тем, чтобы предотвратить переполнение или опустошение буфера, в результате чего могут быть искажены данные или возникнуть угроза безопасности.

Каждый объект памяти также управляет доступом к буферу. Драйвер может выполнять запись только в буфер, предназначенный для получения данных от устройства, такой, как буфер в запросе на чтение. Объект памяти не позволяет драйверу выполнять запись в буфер, предназначенный только для предоставления данных, такой, как буфер в запросе на запись.

Для объектов памяти ведется подсчет ссылок, и они существуют до тех пор, пока не будут удалены все ссылки на них. Но "владельцем" буфера, лежащего в основе объекта памяти, может быть не сам объект памяти. Например, если буфер выделен клиентом, издавшим запрос ввода/вывода, то объект памяти не является "владельцем" буфера. В этом случае указатель на буфер становится недостоверным, когда завершается ассоциированный с ним запрос ввода/вывода, даже если объект памяти продолжает существовать.

Дополнительная информация о времени жизни объектов памяти и лежащих в их основе буферов представлена в *главе 9*.

WDF предоставляет драйверам несколько способов извлечения буферов из запросов ввода/вывода:

- ◆ драйверы UMDF извлекают буфера, вызывая методы интерфейса `IWDFIoRequest` объекта запроса и интерфейса `IWDFMemory` объекта памяти;
- ◆ драйверы KMDF извлекают буфера, вызывая методы `WdfRequestXxx` и `WdfMemoryXxx`.

Общая информация об объектах памяти, интерфейсе `IWDFMemory` и методах семейства `WdfMemoryXxx` приводится в *главе 12*.

### Являются ли объекты памяти необходимыми?

С первого взгляда может казаться, что объекты памяти не являются необходимыми. Почему бы, вместо того, чтобы создавать еще один объект, просто не предоставить прямой указатель на буфер?

В WDM-драйверах запрос может иметь или буфер (для буферизованного ввода/вывода), или список MDL (для прямого ввода/вывода), поэтому драйвер должен удостовериться, что для каждого вызова предоставлен правильный тип. Изменение типа ввода/вывода драйве-

ра WDM с буферизованного на прямой может потребовать капитальной переделки. Что еще хуже, найти все фрагменты кода, требующие модификации, может быть нелегкой задачей, а некоторые могут быть обнаружены только при исполнении драйвера.

Одним из преимуществ объектов памяти WDF является то, что оба типа запросов (буферизованный и прямой) можно абстрагировать, таким образом освобождая драйвер от необходимости использовать специальный код для каждого типа. Если у вас имеется буфер для прямого ввода/вывода, а вам нужен указатель на данные, вызов выполняется в такой же последовательности, как и для буферизованного ввода/вывода. Разницу в обработке WDF берет на себя. В результате, чтобы изменить тип ввода/вывода драйвера WDF с буферизованного на прямой, нужно лишь изменить флаг типа ввода/вывода при инициализации.

Объекты памяти WDF имеют и другие преимущества, которые вы обязательно обнаружите в процессе разработки и отладки ваших драйверов. (Питер Виленд (*Peter Wieland*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Извлечение буферов в драйверах UMDF

В UMDF объект памяти инкапсулирует буфер ввода/вывода, ассоциированный с запросом ввода/вывода. С помощью этого объекта памяти можно копировать данные из драйвера в буфер и в обратном направлении. Объекты памяти предоставляют интерфейс `IWDFMemory`.

Для извлечения объекта памяти из запроса ввода/вывода драйверы применяют методы, перечисленные в табл. 8.3. Эти методы возвращают указатель на интерфейс `IWDFMemory` на объекте.

**Таблица 8.3. Методы интерфейса `IWDFIoRequest` для объектов памяти**

Метод	Описание
<code>GetInputMemory</code>	Извлекает объект памяти, который представляет буфер ввода для запроса ввода/вывода
<code>GetOutputMemory</code>	Извлекает объект памяти, который представляет буфер вывода для запроса ввода/вывода

Получив указатель на интерфейс `IWDFMemory` на объекте памяти, драйвер UMDF может получить информацию о буфере, который описывается объектом памяти, и записывать и считывать данные из этого буфера с помощью методов интерфейса `IWDFMemory`.

Для обращения к буферу драйвер UMDF выполняет следующие шаги:

1. Получает указатель на интерфейс `IWDFMemory`, ассоциированный с запросом. Для этого драйвер вызывает метод `IWDFRequest::GetInputMemory` или `GetOutputMemory` либо оба этих метода, в зависимости от того, является запросом на запись, чтение или IOCTL.
2. Получает указатель на буфер, вызывая для этого метод `IWDFMemory::GetDataBuffer`. Для чтения из буфера драйвер вызывает метод `IWDFMemory::CopyFromBuffer`, а для записи — метод `IWDFMemory::CopyToBuffer`.

Когда драйвер завершает запрос ввода/вывода вызовом метода `IWDFRequest::CompleteXxx`, инфраструктура удаляет объект памяти. После этого указатель на буфер становится недостоверным.

### Внимание!

В настоящее время, независимо от типа запроса IOCTL, драйвер UMDF должен считывать все данные ввода из буфера, перед тем, как записывать данные выхода. Это такое же требование

вание, как и требование к обработке буферизованных запросов IOCTL в драйверах KMDF или WDM.

В листинге 8.1 показано, как образец драйвера Fx2\_Driver извлекает объект памяти и указатель на нижележащий в стеке буфер из объекта запроса ввода/вывода. Этот код является частью кода метода IQueueCallbackDeviceIoControl::OnDeviceIoControl объекта обратного вызова очереди из файла ControlQueue.cpp.

#### Листинг 8.1. Извлечение объекта памяти и указателя на буфер в драйвере UMDF

```
IWDFMemory *memory = NULL;
PVOID buffer;
SIZE_T bigBufferCb;

FxRequest->GetOutputMemory(&memory);
// Получаем адрес буфера, после чего освобождаем объект памяти.
// Объект памяти остается действительным до завершения запроса.
ULONG bufferCb;
buffer = memory->GetDataBuffer(&bigBufferCb);
memory->Release();
if (bigBufferCb > ULONG_MAX) {
    hr = HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER);
    break;
}
else {
    bufferCb = (ULONG) bigBufferCb;
}
```

Чтобы получить указатель на интерфейс IWDFMemory объекта памяти, драйвер вызывает метод IWDFIoRequest::GetOutputMemory на объекте запроса ввода/вывода. Драйвер использует полученный указатель при вызове метода IWDFMemory::GetDataBuffer объекта памяти. Метод GetDataBuffer возвращает в буфер данных значение типа PVOID, которое для доступа к буферу драйвер затем преобразовывает в тип ULONG.

#### Концептуальная инверсия при именовании буферов ввода и вывода

В WDF буферы ввода и вывода рассматриваются таковыми с точки зрения драйвера, а не приложения. Для драйверов WDF буфер в запросе на чтение называется буфером вывода, потому что драйвер выводит в него данные с устройства. А буфер в запросе на запись называется буфером ввода, потому что запрос на запись предоставляет в нем данные для ввода в устройство. Это именование инвертирует стандартное понятие, что чтение представляет операцию ввода, а запись — операцию вывода.

Как для драйверов UMDF, так и для драйверов KMDF методы, возвращающие объекты памяти и буферы для запросов на запись, имеют слово Input в их именах, а методы, возвращающие объекты и буферы для запросов на чтение, имеют в их именах слово Output.

#### Извлечение буферов в драйверах KMDF

В драйверах KMDF объект памяти WDF представляет буфер. С помощью этого объекта памяти можно копировать данные между драйвером и буфером, представленного дескриптором памяти WDF. Кроме этого, драйвер может использовать размер подлежащего буфера и указатель на него для выполнения сложного обращения, например, преобразования в из-

вестный тип структуры данных. В зависимости от типа ввода/вывода, поддерживаемого устройством и драйвером, буфер может быть одним из следующих:

- ◆ для буферизованного ввода/вывода — буфер, выделенный системой из нестраничного пула;
- ◆ для прямого ввода/вывода — выделенный системой список MDL, указывающий на физические страницы для DMA;
- ◆ для ввода/вывода типа METHOD\_NEITHER, запрошенного клиентом пользовательского режима — буфер пользовательского режима, проверенный на достоверность и отображеный и заблокированный в виртуальном адресном пространстве ядра методами `WdfRequestRetrieveUnsafeUserXxxBuffer` и `WdfRequestProbeAndLockUserBufferForXxx`.

Методы семейства `WdfRequestRetrieveXxx` возвращают объекты памяти и буфера из запросов. Краткое описание этих методов приводится в табл. 8.4.

**Таблица 8.4. Методы семейства `WdfRequestRetrieveXxx`**

Метод	Описание
<code>WdfRequestRetrieveInputBuffer</code>	Извлекает буфер ввода запроса ввода/вывода
<code>WdfRequestRetrieveInputMemory</code>	Извлекает дескриптор объекта типа <code>WDFMEMORY</code> , представляющего буфер ввода запроса ввода/вывода
<code>WdfRequestRetrieveInputWdmMdl</code>	Извлекает список MDL, представляющий буфер ввода запроса ввода/вывода
<code>WdfRequestRetrieveOutputBuffer</code>	Извлекает буфер вывода запроса ввода/вывода
<code>WdfRequestRetrieveOutputMemory</code>	Извлекает дескриптор объекта типа <code>WDFMEMORY</code> , представляющего буфер вывода запроса ввода/вывода
<code>WdfRequestRetrieveOutputWdmMdl</code>	Извлекает список MDL, представляющий буфер вывода запроса ввода/вывода
<code>WdfRequestRetrieveUnsafeUserInputBuffer</code>	Извлекает буфер ввода запроса ввода/вывода для запроса типа <code>METHOD_NEITHER</code>
<code>WdfRequestRetrieveUnsafeUserOutputBuffer</code>	Извлекает буфер вывода запроса ввода/вывода для запроса типа <code>METHOD_NEITHER</code>

Имея дескриптор объекта типа `WDFMEMORY`, драйвер может манипулировать этим объектом и выполнять операции чтения и записи с его буферами, применяя методы семейства `WdfMemoryXxx`. Если драйвер исполняет буферизованный или прямой ввод/вывод, он может обращаться к буферам любым из следующих способов.

- ◆ Получить дескриптор объекта типа `WDFMEMORY`, ассоциированного с `WDFREQUEST`. Для этого применяется метод `WdfRequestRetrieveInputMemory` для запросов на запись и метод `WdfRequest` для запросов на чтение. После этого получить указатель на буфер, вызвав метод `WdfMemoryGetBuffer`.
- ◆ Теперь драйвер может выполнять операции чтения и записи с буфером с помощью методов `WdfMemoryCopyFromBuffer` и `WdfMemoryCopyToBuffer` соответственно.
- ◆ Получить указатель на буфер. Это делается с помощью вызова метода `WdfRequestRetrieveInputBuffer` для запросов на запись и метода `WdfRequestRetrieveOutputBuffer` для запросов на чтение.

Драйвер, выполняющий прямой ввод/вывод, может также обращаться к лежащему в основе буфера списку MDL, вызывая для этого методы `WdfRequestRetrieveInputWdmMdl` и `WdfRequestRetrieveOutputWdmMdl`. Но драйверу нельзя применять адрес буфера пользовательского режима, указанный в списке MDL; вместо этого он должен с помощью макроса `MmGetSystemAddressForMdlSafe` получить адрес режима ядра. Методы `WdfRequestRetrieveInputBuffer` и `WdfRequestRetrieveOutputBuffer` проще в использовании, т. к. инфраструктура извлекает адрес режима ядра для драйвера на основе типа буфера запроса.

Для запросов ввода/вывода типа `METHOD_NEITHER` KMDF предоставляет методы для анализа и блокирования буферов пользовательского режима. Для анализа и блокировки буфера пользовательского режима драйвер должен исполняться в контексте процесса, издавшего запрос ввода/вывода. Поэтому KMDF также определяет функцию обратного вызова, которую драйверы могут зарегистрировать для вызова в контексте посылающего компонента. Для получения дополнительной информации по этому вопросу см. разд. "Обращение к буферам в драйверах KMDF при вводе/выводе типа `METHOD_NEITHER`" далее в этой главе.

### Обращение к буферам при буферизованном и прямом вводе/выводе в драйверах KMDF

Если драйвер KMDF поддерживает буферизованный или прямой ввод/вывод, он может выполнять операции чтения и записи или с буфером, переданным инициатором запроса ввода/вывода, или с объектом типа `WDFMEMORY`, представляющим этот буфер.

Все вопросы адресации и проверки достоверности для объектов типа `WDFMEMORY` улаживаются инфраструктурой, которая также не допускает выполнения драйвером операций записи в буфер ввода. Дескриптор объекта типа `WDFMEMORY` содержит размер буфера, таким образом предотвращая переполнение буфера. Поэтому при применении объектов типа `WDFMEMORY` обычно требуется меньший объем кода. Если драйвер выполняет операции чтения и записи с буфером посредством указателей на буфер, инфраструктура выполняет первоначальную проверку достоверности размеров буферов, но не предотвращает переполнение или опустошение буфера, когда драйвер копирует данные в буфер.

Чтобы получить дескриптор объекта `WDFMEMORY`, драйвер вызывает:

- ◆ метод `WdfRequestRetrieveOutputMemory` для запроса на чтение;
- ◆ метод `WdfRequestRetrieveInputMemory` для запроса на запись;
- ◆ оба предыдущих метода для запросов IOCTL.

Все эти методы возвращают дескриптор объекта типа `WDFMEMORY`, который представляет соответствующий буфер и ассоциирован с объектом типа `WDFREQUEST`.

В альтернативе, драйвер может получить указатели на сами буфера ввода и вывода с помощью методов `WdfRequestRetrieveOutputBuffer` и `WdfRequestRetrieveInputBuffer`.

Для буферизованных запросов ввода/вывода, как для ввода, так и для вывода, применяется тот же самый буфер, так что оба эти метода возвращают тот же самый указатель. Поэтому, при выполнении операций ввода/вывода буферизованного типа, драйвер сначала должен прочитать и сохранить все входные данные из буфера перед тем, как записывать в него выходные данные.

Драйвер Osrusbfx2 выполняет буферизованный ввод/вывод. Когда этот драйвер получает запрос на чтение, он извлекает объект типа `WDFMEMORY` из запроса с помощью следующего оператора:

```
status = WdfRequestRetrieveOutputMemory(Request, &reqMemory);
```

Этот вызов метода `WdfRequestRetrieveOutputMemory` возвращает дескриптор объекта памяти по адресу `reqMemory`. Драйвер потом проверяет достоверность запроса и посыпает его каналу USB получателя.

### **Обращение к буферам в драйверах KMDF при вводе/выводе типа METHOD\_NEITHER**

Драйвер KMDF, выполняющий ввод/вывод типа `METHOD_NEITHER`, получает указатель на буфер пользовательского режима для запросов, инициированных клиентом пользовательского режима. Прежде чем осуществлять доступ к нему, драйвер должен проверить достоверность этого указателя и заблокировать память буфера пользовательского режима. После этого драйвер должен сохранить указатель на буфер в области контекста объекта запроса для дальнейшего использования при обработке и завершении запрошенного ввода/вывода. Но обратите внимание, что это требование не распространяется на запросы, исходящие от клиентов режима ядра.

Чтобы проверить достоверность указателя на буфер и заблокировать память буфера, драйвер должен исполняться в контексте процесса пользовательского режима, инициировавшего запрос. Чтобы обеспечить вызов драйвера в контексте пользователя, драйвер KMDF, выполняющий ввод/вывод типа `METHOD_NEITHER`, должен реализовать обратный вызов функции `EvtIoInCallerContext` и зарегистрировать этот обратный вызов с помощью метода `WdfDeviceInitSetIoInCallerContextCallback` при инициализации объекта устройства. Инфраструктура вызывает функцию `EvtIoInCallerContext` в контексте вызывающего клиента каждый раз, когда она получает запрос ввода/вывода для объекта устройства. Эта функция должна получить из запроса указатель на буфер, проверить его достоверность и заблокировать память буфера пользовательского режима.

Функция `EvtIoInCallerContext` может получить буферы из запросов ввода/вывода типа `METHOD_NEITHER`, вызвав один из следующих методов:

- ◆ метод `WdfRequestRetrieveUnsafeUserInputBuffer` получает буфер для запросов на чтение или IOCTL;
- ◆ метод `WdfRequestRetrieveUnsafeUserOutputBuffer` получает буфер для запросов на запись или IOCTL.

Эти методы проверяют достоверность размеров буферов, поставляемых в запросе, и возвращают указатель на буфер пользовательского режима. Если размер буфера недействителен, или тип ввода/вывода не является `METHOD_NEITHER`, или инфраструктура обнаружит какие-либо другие ошибки, эти методы возвращают статус неуспешного завершения. Они должны вызываться только из функции обратного вызова `EvtIoInCallerContext` драйвера и завершаться неудачей, если вызваны из любой другой функции обратного вызова.

Прежде чем обращаться к буферу, драйвер должен удостовериться, что буферы действительны для операций чтения и записи, и заблокировать их страницы физической памяти. Эта задача выполняется с помощью методов `WdfRequestProbeAndLockUserBufferForRead` и `WdfRequestProbeAndLockUserBufferForWrite`. В терминах WDM эти методы выполняют следующие действия.

- ◆ Выделяют список MDL для буфера пользовательского режима.
- ◆ Вызывают процедуру `MmProbeAndLockPages` для проверки достоверности буфера пользовательского режима и блокировки его физической памяти. Буферы автоматически разблокируются инфраструктурой по завершению запроса.

- ◆ Вызывают макрос `MmGetSystemAddressForMdlSafe`, чтобы отобразить адрес пользовательского режима в адресное пространство режима ядра.
- ◆ Создают объект памяти WDF для представления буфера.
- ◆ Добавляют этот объект памяти WDF к дочерним объектам объекта запроса WDF.
- ◆ Устанавливают объект памяти WDF указывать на ранее выделенный список MDL.

После того как драйвер заблокирует физические страницы памяти буфера, он должен сохранить указатель на буфер в области контекста объекта типа `WDFREQUEST`. Если заблокированный буфер содержит встроенные указатели, функция обратного вызова `EvtIoInCallerContext` должна проверить достоверность этих указателей таким же образом. Если буфер пользовательского режима содержит входные данные, драйвер должен сохранить эти данные в безопасном буфере режима ядра.

Далее приводится код из образца драйвера `Nopprp`, который использует ввод/вывод типа `METHOD_NEITHER` для запроса `IOCTL`. В нем показано, как создается область контекста объекта для запроса, сохраняются указатели на буфера и возвращается запрос инфраструктуре для постановки в очередь. Создавая новую область контекста для этого запроса, вместо того чтобы ассоциировать область контекста с каждым запросом ввода/вывода, создаваемым инфраструктурой, драйвер обеспечивает, что память используется только для тех запросов, которые требуют этого.

В заголовочном файле драйвер обявляет тип контекста и именует функцию доступа следующим образом:

```
typedef struct _REQUEST_CONTEXT {
    WDFMEMORY InputMemoryBuffer;
    WDFMEMORY OutputMemoryBuffer;
} REQUEST_CONTEXT, *PREQUEST_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(REQUEST_CONTEXT, CetRequestContext)
```

В своей функции обратного вызова `EvtIoInCallerContext` драйвер выделяет область контекста для этого запроса, проверяет достоверность буферов ввода и вывода и блокирует их страницы физической памяти, сохраняет указатели на буфера в области контекста, после чего возвращает запрос инфраструктуре для постановки в очередь. В листинге 8.2 приводится исходный код функции `EvtIoInCallerContext` драйвера из файла `Nopprp.c`.

#### Листинг 8.2. Функция обратного вызова `EvtIoInCallerContext`

```
VOID NonPnpEvtDeviceIoInCallerContext(
    IN WDFDEVICE Device,
    IN WDFREQUEST Request)
{
    NTSTATUS status = STATUS_SUCCESS;
    PREQUEST_CONTEXT reqContext = NULL;
    WDF_OBJECT_ATTRIBUTES attributes;
    size_t inBufLen, outBufLen;
    PVOID inBuf, outBuf;

    . . . // Код опущен.

    // Выделяется область контекста для этого запроса для хранения
    // объектов памяти, созданных для буферов ввода и вывода.
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, REQUEST_CONTEXT);
    status = WdfObjectAllocateContext(Request, &attributes, &reqContext);
```

```

if (!NT_SUCCESS(status)) {
    goto End;
}
status = WdfRequestProbeAndLockUserBufferForRead(Request,
    inBuf,
    inBufLen,
    &reqContext->InputMemoryBuffer);
if (!NT_SUCCESS(status)) {
    goto End;
}
status = WdfRequestProbeAndLockUserBufferForWrite(Request,
    outBuf,
    outBufLen,
    &reqContext->OutputMemoryBuffer);
status = WdfDeviceEnqueueRequest(Device, Request);
if (!NT_SUCCESS(status)) {
    goto End;
}
return;
End:
WdfRequestComplete(Request, status);
return;
}

```

Метод `WdfDeviceEnqueueRequest` вынуждает инфраструктурный обработчик ввода/вывода поставить запрос в очередь в обычном порядке, как показано на рис. 8.5.

Если драйвер сконфигурировал параллельную или последовательную диспетчеризацию для очереди, инфраструктура позже активирует обратный вызов по событию ввода/вывода для типа ввода/вывода, указанного в запросе.

Если драйвер сконфигурировал очередь для ручной диспетчеризации, он вызывает один из методов семейства `WdfIoQueueRetrieveXxx`, когда готов получить и обработать запрос. Прежде чем обратиться к буферу, драйвер использует функцию доступа для объекта типа `WDFREQUEST` (в примере это функция `GetRequestContext`), чтобы получить указатель на область контекста, из которой он может извлечь указатель на буфер.

Так как запрос пока еще не был поставлен в очередь, его можно отменить пока исполняется функция обратного вызова `EvtIoInCallerContext`. Обычно драйвер просто блокирует страницы памяти буфера, сохраняет указатель и возвращает запрос инфраструктуре. Если в это время запрос отменяется, это не затрагивает драйвер, а инфраструктура просто завершает запрос позже, возвращая статус `STATUS_CANCELLED`. По завершению запроса инфраструктура отменяет отображение памяти буферов на адресное пространство режима ядра, выполненное драйвером с помощью методов `WdfRequestProbeAndLockUserBufferXxx`. Выполнять очистку в драйвере не требуется.

Но если драйвер выделяет ресурсы, связанные с запросом, или выполняет другие действия, требующие очистки при отмене запроса, драйвер должен зарегистрировать обратный вызов функции `EvtObjectCleanup` для объекта запроса WDF.

Если запрос отменяется после того, как драйвер вызвал метод `WdfDeviceEnqueueRequest`, инфраструктура активирует обратный вызов функции `EvtIoCanceledOnQueue`, если такой был зарегистрирован драйвером. Драйвер должен зарегистрировать эту функцию обратного вызова, если он должен выполнить очистку, связанную с обслуживанием запроса, выполненно-

го после постановки его в очередь драйвером. Если драйвер регистрирует функцию обратного вызова `EvtIoCanceledOnQueue`, обычно он также регистрирует обратный вызов функции `EvtObjectCleanup` для объекта запроса WDF. Функции обратного вызова `EvtIoCanceledOnQueue` и `EvtObjectCleanup` требуются только в том случае, если драйвер выделяет ресурсы, связанные с запросом, или выполняет другие действия, требующие очистки при отмене запроса.

В листинге 8.2 функция `EvtIoInCallerContext` вызывает метод `WdfDeviceEnqueueRequest`, чтобы возвратить запрос инфраструктуре для постановки в очередь. Если драйвер не возвращает запрос инфраструктуре для постановки в очередь, он должен быть способным выполнить отмену запроса, а также предоставить любую требуемую для этого синхронизацию. Обычно вместо того, чтобы пытаться реализовывать собственные возможности постановки в очередь и управления очередями, драйверы должны пользоваться возможностями для этого, предоставляемыми инфраструктурой.

## Время жизни запросов, памяти и указателей на буферы

Как было упомянуто ранее, время жизни объекта памяти связано со временем жизни объекта запроса ввода/вывода, являющегося его родителем. Но время жизни указателя на подлежащий буфер может быть другим.

При самом простом сценарии инфраструктура направляет запрос драйверу, который выполняет ввод/вывод и завершает запрос. В данном случае подлежащий буфер мог быть создан приложением пользователяского режима, другим драйвером или самой операционной системой. Когда драйвер завершает запрос ввода/вывода, инфраструктура удаляет объекты памяти, ассоциированные с запросом. После этого указатель на буфер становится недействительным.

Если драйвер создает буфер, объект памяти или запрос или если драйвер пересыпает запрос получателю ввода/вывода, вопрос времени жизни усложняется.

Подробности о времени жизни объектов памяти и указателей на буферы в запросах, посылаемых драйвером получателю ввода/вывода, предоставлены в главе 9.

## Очереди ввода/вывода

Объект очереди ввода/вывода представляет очередь, которая предоставляет запросы ввода/вывода драйверу и способ управления потоком запросов ввода/вывода к драйверу. Но очередь ввода/вывода WDF — это больше, чем просто список запросов, ожидающих выполнения. Объекты очереди отслеживают в драйвере активные запросы, поддерживают отмену запросов и управляют параллельным исполнением запросов. Кроме этого, они дополнительно могут синхронизировать свои действия с состоянием Plug and Play и энергопотребления устройства, а также синхронизировать обратные вызовы функций драйвера по событиям ввода/вывода.

Драйвер создает свои очереди ввода/вывода после создания объекта устройства. Для этого он вызывает один из следующих методов:

- ◆ драйвер UMDF вызывает метод `IWDFDevice::CreateIoQueue`;
- ◆ драйвер KMDF вызывает метод `WdfIoQueueCreate`.

Для каждого объекта устройства драйвер обычно создает одну или несколько очередей. Каждая очередь может принимать один или несколько типов запросов. Для каждой очереди драйвер может указать следующее:

- ◆ тип запросов, которые ставятся в данную очередь;
- ◆ опции управления энергопотреблением для данной очереди;
- ◆ метод отправки запросов, который определяет количество разрешенных активных запросов в драйвере в любой данный момент;
- ◆ принимает ли очередь запросы с нулевым размером буфера.

Пока запрос находится в очереди и еще не передан драйверу, его владельцем считается очередь. После того как запрос отправляется драйверу, его владельцем становится драйвер, а запрос считается находящимся в транзите. Внутренне, каждая очередь отслеживает, какими запросами она владеет, а какие ожидают исполнения. Драйвер может передавать запросы из одной очереди в другую, вызывая соответствующий метод объекта.

### **Очереди как компоновочные блоки**

Очереди играют роль автономных объектов и компоновочных блоков для сортировки запросов и реализации более сложных моделей маршрутизации. При объяснении разработчикам сути очереди, мы очень часто применяем термин "компонентные блоки", и я думаю, что этот термин хорошо выражает наши намерения относительно применения очереди разработчиками. (Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## **Конфигурация очереди и типы запросов**

Драйвер может иметь любое количество очередей, каждая из которых может быть сконфигурирована по-своему. Например, драйвер может иметь отдельные очереди для запросов на чтение, запись и IOCTL, и для каждой из этих очередей может применяться свой вид диспетчеризации. Драйвер также может создавать очереди для своего внутреннего пользования.

Не все типы запросов ввода/вывода можно поставить в очередь. Инфраструктура направляет некоторые типы запросов драйверам сразу же, активируя функцию обратного вызова. В табл. 8.5 приведены типы запросов ввода/вывода, которые инфраструктура ставит в очередь, и типы, которые она отправляет функциям обратного вызова без постановки в очередь.

**Таблица 8.5. Механизм доставки запросов ввода/вывода**

Тип запроса ввода/вывода	Механизм доставки UMDF	Механизм доставки KMDF
На чтение	Постановка в очередь	Постановка в очередь
На запись	Постановка в очередь	Постановка в очередь
IOCTL	Постановка в очередь	Постановка в очередь
Внутренний IOCTL	Постановка в очередь	Постановка в очередь
На создание	Постановка в очередь	Постановка в очередь или обратный вызов
На закрытие	Обратный вызов	Обратный вызов
На очистку	Обратный вызов	Обратный вызов

Хотя UMDF не ставит запросы на очистку или отмену в формальную очередь, некоторое ожидание может иметь место на внутреннем уровне. Количество потоков, которым инфраструктура может доставить запросы ввода/вывода, ограничено, так что существует возможность задержки запросов на очистку или закрытие, пока инфраструктура ожидает свободного потока.

## Указание типа запроса для очереди

Драйвер указывает тип запросов ввода/вывода, принимаемых очередью, следующим образом.

- ◆ Драйвер UMDF вызывает либо метод `IWDFDevice::ConfigureRequestDispatching` объекта устройства, либо метод `IWDFIoQueue::ConfigureRequestDispatching` объекта очереди.

В вызове метода `ConfigureRequestDispatching` драйвер предоставляет тип запроса ввода/вывода и булево значение, указывающее, должна ли инфраструктура ставить в очередь запросы этого типа. Чтобы сконфигурировать все типы запросов для очереди, драйвер может вызывать метод `ConfigureRequestDispatching` необходимое количество раз. Действительными типами запросов являются следующие:

- `WdfRequestCreate;`
- `WdfRequestRead;`
- `WdfRequestWrite;`
- `WdfRequestDeviceIoControl.`

Драйвер также должен реализовать соответствующие интерфейсы обратного вызова очереди на объекте обратного вызова очереди. Чтобы определить интерфейсы, поддерживаемые объектом очереди, инфраструктура вызывает метод `QueryInterface` объекта обратного вызова.

- ◆ Для каждого типа запроса ввода/вывода, поддерживаемого очередью, драйвер KMDF вызывает метод `WdfDeviceConfigureRequestDispatching`. Действительными типами запросов являются следующие:

- `WdfRequestTypeCreate;`
- `WdfRequestTypeRead;`
- `WdfRequestTypeWrite;`
- `WdfRequestTypeDeviceControl;`
- `WdfRequestTypeDeviceControlInternal.`

Для каждой очереди драйвер также должен зарегистрировать функции обратного вызова по событиям ввода/вывода, устанавливая соответствующие поля в структуре конфигурации очереди. Таким образом выполняется инициализация этой структуры, показано в разд. "Пример KMDF: создание очередей ввода/вывода" далее в этой главе.

## Обратные вызовы для очереди

В табл. 8.6 перечислены интерфейсы и функции обратного вызова по событию, которые драйвер может реализовать для очереди ввода/вывода.

**Таблица 8.6.** Обратные вызовы для очередей ввода/вывода

Ассоциированное событие	Интерфейс обратного вызова UMDF	Функция обратного вызова KMDF
Запрос на чтение	IQueueCallbackRead	EvtIoRead
Запрос на запись	IQueueCallbackWrite	EvtIoWrite
Запрос IOCTL	IQueueCallbackDeviceIoControl	EvtIoDeviceIoControl
Внутренний запрос IOCTL	Не применимо	EvtIoInternalDeviceIoControl
Запрос на создание	IQueueCallbackCreate	EvtIoDefault
Запрос ввода/вывода, для которого не реализовано никаких других обратных вызовов	IQueueCallbackDefaultIoHandler	EvtIoDefault
Извещение об остановке управляемой энергопотреблением очереди	IQueueCallbackIoStop	EvtIoStop
Извещение о возобновлении работы управляемой энергопотреблением очереди	IQueueCallbackIoResume	EvtIoResume
Извещение об изменении в состоянии очереди	IQueueCallbackStateChange	EvtIoQueueState
Извещение об отмене поставленного в очередь запроса	Не применимо	EvtIoCanceledOnQueue

Как можно видеть в предыдущей таблице, в добавок к запросам ввода/вывода, очередь может поддерживать обратные вызовы функций для следующих типов событий:

- ◆ изменение управления энергопотреблением, как описано в разд. "Управляемые энергопотреблением очереди" далее в этой главе;
- ◆ изменение в состоянии очереди, как описано в разд. "Управление очередью" далее в этой главе;
- ◆ для драйверов KMDF, отмена запроса ввода/вывода, как описано в разд. "Отмененные и приостановленные запросы" далее в этой главе.

## Стандартные очереди

Для каждого объекта устройства можно создать и сконфигурировать одну стандартную очередь, куда инфраструктура помещает запросы, для которых драйвер специфически не сконфигурировал других очередей. Если объект устройства имеет стандартную очередь и одну или несколько специфичных очередей, инфраструктура помещает специфичные запросы в соответствующие очереди, а все остальные запросы — в стандартную очередь. Для стандартной очереди драйвер не вызывает инфраструктурный метод для конфигурирования метода диспетчеризации запросов.

Если объект устройства не имеет стандартной очереди, инфраструктура ставит в очередь только указанные типы запросов в соответственно сконфигурированные очереди, а остальные запросы обрабатывает стандартным образом, зависящим от типа драйвера. Например, для функционального драйвера и драйвера шины инфраструктура завершает неудачей неуказанные типы запросов. А для драйвера фильтра другие типы запросов передаются следующему нижележащему драйверу.

### Примечание

Драйверу не обязательно иметь стандартную очередь ввода/вывода. Например, драйвер, который обрабатывает только запросы на чтение и IOCTL, может сконфигурировать одну очередь для запросов на чтение, а другую — для запросов IOCTL. В этом сценарии, если драйвер получает запрос на запись, его дальнейшая часть зависит от типа драйвера. Для функционального драйвера инфраструктура завершает запрос неудачей, а для драйвера фильтра — передает его вниз по стеку стандартному получателю ввода/вывода.

## Очереди и управление энергопотреблением

Внутренне WDF объединяет поддержку очередей с машиной состояний Plug and Play и управления энергопотреблением. В результате WDF может синхронизировать операции очередей с состоянием энергопотребления устройства или же драйвер может управлять очередями самостоятельно. Управление энергопотреблением можно конфигурировать для каждой очереди. Драйвер может использовать как управляемые, так и неуправляемые энергопотреблением очереди и способен сортировать запросы на основе требований его модели энергопотребления.

### Совет

Управляемые энергопотреблением очереди следует применять для таких запросов, которые драйвер может обрабатывать только тогда, когда устройство пребывает в рабочем состоянии. Неуправляемые энергопотреблением очереди следует применять для таких запросов, которые драйвер может обрабатывать даже тогда, когда устройство не находится в рабочем состоянии.

### Управляемые энергопотреблением очереди

По умолчанию для функциональных драйверов и драйверов шины применяются управляемые энергопотреблением очереди. Это означает, что состояние очереди активирует деятельность управления энергопотреблением и наоборот. Управляемые энергопотреблением очереди дают драйверам следующие преимущества.

- ◆ Если запрос ввода/вывода прибывает, когда система находится в рабочем состоянии (S0), но устройство не находится в рабочем состоянии, обработчик ввода/вывода дает указание обработчикам Plug and Play и энергопотребления восстановить рабочее питание устройству.
- ◆ Драйвер может реализовать для очереди функцию обратного вызова по событию остановки ввода/вывода.

Если в результате изменения состояния устройства очередь останавливается, инфраструктура активирует обратный вызов функции для каждого запроса ввода/вывода, принадлежащего драйверу, который был отправлен очередью. В функции обратного вызова драйвер может завершить, отменить или подтвердить запрос ввода/вывода до того, как устройство выйдет из рабочего состояния.

- ◆ Применительно только к KMDF, при освобождении очереди инфраструктурный обработчик ввода/вывода извещает обработчика событий Plug and Play и энергопотребления, с тем, чтобы последний начал отслеживать время простоя устройства, запустив таймер периода простоя. Если устройство поддерживает переход в состояние пониженного энергопотребления при простое, когда значение таймера периода простоя примет нулевое значение, обработчик ввода/вывода может отменить запрос ввода/вывода.

вое значение, обработчик событий Plug and Play и энергопотребления может перевести устройство в одно из состояний пониженного энергопотребления.

- ◆ Когда устройство покидает рабочее состояние (D0), инфраструктура приостанавливает доставку запросов и возобновляет ее по возвращению устройства в рабочее состояние. Хотя при остановке очереди отправление запросов из очереди прекращается, постановка запросов в очередь продолжается.

Чтобы очередь доставляла запросы, как драйвер, так и состояние энергопотребления устройства должны разрешать обработку.

Состояние очереди и действия инфраструктуры описаны в табл. 8.7.

**Таблица 8.7. "Взаимоотношения" очереди и инфраструктуры**

Состояние очереди при прибытии запроса	Действие инфраструктуры
Очередь отправляет запросы	Добавляет запрос в очередь для доставки в соответствии с типом диспетчеризации очереди
Очередь остановлена	Добавляет запрос в очередь для доставки, когда очередь возобновит работу
Очередь остановлена, устройство находится в состоянии пониженного энергопотребления, а система пребывает в рабочем состоянии	Добавляет запрос в очередь и возвращает устройство в рабочее состояние перед тем, как доставить запрос
Очередь остановлена и система находится в процессе перехода в режим пониженного энергопотребления	Добавляет запрос в очередь и после возвращения системы в рабочее состояние возвращает устройство в рабочее состояние

Дополнительно, драйвер может реализовать обратные вызовы функций по событию ввода/вывода, которые система активирует, когда управляемая энергопотреблением очередь останавливается или возобновляет работу в связи с изменением в состоянии энергопотребления устройства. Инфраструктура вызывает эти функции по одному разу для каждого запроса, принадлежащего драйверу. Эти функции обратного вызова перечислены в табл. 8.8.

**Таблица 8.8. Функции обратного вызова по событию ввода/вывода для изменений в состоянии очереди**

Если драйвер UMDF	Или если драйвер KMDF	Инфраструктура вызывает драйвер
Реализует <code>IQueueCallbackIoStop</code>	Зарегистрировал <code>EvtIoStop</code>	Перед тем, как устройство покинет рабочее состояние, чтобы драйвер мог приостановить обработку запроса или удалить и завершить запрос
Реализует <code>QueueCallbackIoResume</code>	Зарегистрировал <code>EvtIoResume</code>	После возвращения устройства в рабочее состояние. Но KMDF вызывает драйвер, только если функция обратного вызова драйвера <code>EvtIoStop</code> подтвердила остановку во время перехода устройства в состояние пониженного энергопотребления, но не поставила запрос в очередь снова

Дополнительная информация о функциях обратного вызова для остановок ввода/вывода приводится в разд. "Приостановка запроса ввода/вывода" далее в этой главе.

## Неуправляемые энергопотреблением очереди

Очереди, не управляемые энергопотреблением, доставляют запросы ввода/вывода драйверу до тех пор, пока устройство присутствует, независимо от его состояния энергопотребления. Если устройство находится в состоянии пониженного энергопотребления, когда прибывает запрос ввода/вывода, WDF не переводит устройство в рабочее состояние. Неуправляемые энергопотреблением очереди следует применять для таких запросов, которые драйвер может обрабатывать даже тогда, когда его устройство не находится в рабочем состоянии.

Таймеры простоя устройства запускаются только для управляемых энергопотреблением очередей. Если драйвер поддерживает перевод простоявшего устройства в состояние пониженного энергопотребления, состояние неуправляемой энергопотреблением очереди не влияет на решение инфраструктуры о запуске или остановке таймера периода простоя устройства. Драйвер может программным образом перевести устройство в рабочее состояние или запретить ему переходить в состояние пониженного энергопотребления, вызвав метод `WdfDeviceStopIdle`.

Для неуправляемых энергопотреблением очередей драйвер может реализовать обратный вызов функции по событию остановки ввода/вывода, как описано в табл. 8.8. Инфраструктура активирует этот обратный вызов только при удалении устройства, но не тогда, когда устройство переходит в состояние пониженного энергопотребления или останавливается для перераспределения ресурсов.

## Тип диспетчеризации

Тип диспетчеризации очереди определяет, как и когда запросы ввода/вывода доставляются драйверу, и, в результате, могут ли несколько запросов ввода/вывода из очереди быть одновременно активными в драйвере. Драйверы могут управлять параллелизмом запросов, находящихся в транзите, конфигурируя тип диспетчеризации для своих очередей. WDF поддерживает следующие три типа диспетчеризации.

- ◆ **Последовательная.** Очередь отправляет драйверу запросы ввода/вывода по одному. Очередь не отправляет драйверу новый запрос до тех пор, пока предыдущий не был завершен или переслан другой очереди. Если драйвер послал запрос получателю ввода/вывода, очередь не посыпает другой запрос до тех пор, пока текущий драйвер не завершит запрос. Последовательная диспетчеризация похожа на метод "start I/O" в WDM.
- ◆ **Параллельная.** Очередь посыпает драйверу запросы ввода/вывода, как только это становится возможным, независимо от того, обрабатывает ли драйвер в это время другой запрос.
- ◆ **Ручная.** Драйвер извлекает запросы из очереди самостоятельно, когда ему это нужно.

Все запросы ввода/вывода, получаемые драйвером из очереди, по существу являются асинхронными. Драйвер может завершить запрос в соответствующей функции обратного вызова по событию ввода/вывода или же некоторое время позже, после возвращения управления функцией обратного вызова. Драйвер не обязан помечать запрос как незаконченный, как это должны делать драйверы WDM; это за него делает WDF.

Тип диспетчеризации управляет только количеством одновременных активных запросов в драйвере. На тип вызовов функций обратного вызова очереди для событий ввода/вывода — последовательный или параллельный — тип диспетчеризации не влияет. Вопрос паралле-

лизма обратных вызовов решается с помощью области синхронизации объекта устройства. Но даже если область синхронизации для параллельной очереди не позволяет параллельных обратных вызовов, очередь, тем не менее, может иметь несколько запросов в транзите одновременно.

### **Запросы, ожидающие завершения и статус функций обратного вызова для событий ввода/вывода**

Если вы знакомы с драйверами WDM, то, может быть, удивляйтесь, почему инфраструктура не имеет метода для обозначения запросов как незаконченные, и почему функции обратного вызова драйвера для событий ввода/вывода не возвращают статуса.

Инфраструктура позволила избавиться от всех правил и сложности в драйверах, связанных с необходимостью обозначать пакет IRP как незаконченный и возвращать статус STATUS\_PENDING. Внутренне инфраструктура всегда обозначает пакет IRP как незаконченный и возвращает статус STATUS\_PENDING даже если процедура драйвера завершает запрос в это же время. Менеджер ввода/вывода оптимизирует обработку ввода/вывода в зависимости от контекста потока, поэтому обозначение запроса незаконченным и завершение его в это же время не влияют на производительность. Это также позволило нам избавиться от возвращения статуса функциями обратного вызова для событий ввода/вывода драйвера, т. к. это не служила никакой определенной цели. Когда драйвер завершает запрос, инфраструктура завершает пакет IRP. (*Ильяс Якуб (Eliyas Yakub), команда разработчиков Windows Driver Foundation, Microsoft.*)

## **Управление очередью**

Инфраструктура WDF предоставляет методы, с помощью которых драйвер может остановить, запустить, очистить<sup>1</sup> (drain) и удалить<sup>2</sup> (purge) очереди ввода/вывода. А именно:

- ◆ драйверы UMDF управляют очередями ввода/вывода, вызывая методы интерфейса IWDFIoQueue объекта очереди;
- ◆ драйверы KMDF управляют очередями ввода/вывода, вызывая методы семейства WdfIoQueueXxx.

WDF поддерживает как синхронные, так и асинхронные методы остановки, удаления и очистки очередей. Синхронные методы возвращают управления после завершения операции. Операция очистки или удаления завершается, когда завершены все запросы в очереди.

Асинхронные методы возвращают управление немедленно и активируют предоставляемую драйвером функцию обратного вызова по завершению операции. Чтобы зарегистрировать эту функцию обратного вызова:

- ◆ драйвер UMDF предоставляет указатель на интерфейс IQueueCallbackStateChange объекта обратного вызова очереди при вызове асинхронного метода управления очередью;
- ◆ драйвер KMDF предоставляет указатель на функцию EvtIoQueueState обратного вызова при вызове асинхронного метода управления очередью.

Методы управления очередями приводятся в табл. 8.9.

<sup>1</sup> Очистка (drain) очереди означает, что очередь не принимает новые входящие запросы ввода/вывода, но доставляет уже поставленные в очередь запросы драйверу для обработки. — *Пер.*

<sup>2</sup> Удаление (purge) очереди означает, что очередь не принимает новых запросов, и все незаконченные запросы отменяются. — *Пер.*

Таблица 8.9. Методы управления очередями

Операция	Метод UMDF интерфейса IWDFIoQueue	Метод KMDF
Останавливает добавление запросов в очередь.  Дополнительно извещает драйвер или возвращает управление драйверу после завершения всех незаконченных запросов ввода/вывода	Drain  DrainSynchronously	WdfIoQueueDrain  WdfIoQueueDrainSynchronously
Возвращает состояние очереди ввода/вывода	GetState	WdfIoQueueGetState
Останавливает добавление запросов в очередь.  Отменяет все запросы, стоящие в очереди, и все отменяемые запросы, находящиеся в транзите.  Извещает драйвер или возвращает управление драйверу только после завершения всех незаконченных запросов ввода/вывода	Purge  PurgeSynchronously	WdfIoQueuePurge  WdfIoQueuePurgeSynchronously
Возобновляет доставку запросов из очереди драйверу	Start	WdfIoQueueStart
Останавливает доставку запросов из очереди драйверам, но продолжает добавлять новые запросы в очередь	Stop  StopSynchronously	WdfIoQueueStop  WdfIoQueueStopSynchronously

Драйвер может использовать методы управления очередями совместно с обратными вызовами для самоуправляемого ввода/вывода, чтобы управлять вручную неуправляемыми энергопотреблением очередями. Поддержка самоуправляемого ввода/вывода в WDF описывается в разд. "Самоуправляемый ввод/вывод" далее в этой главе.

## Пример UMDF: создание очередей ввода/вывода

Чтобы создать очередь ввода/вывода, драйвер UMDF выполняет такую последовательность действий:

- Если драйвер реализует интерфейсы обратных вызовов для очереди, создает объект обратного вызова очереди.
- Создает инфраструктурный объект очереди, вызывая для этого метод `IWDFDevice::CreateIoQueue`.
- Конфигурирует объект очереди для принятия одного или нескольких типов запросов ввода/вывода (если только очередь не является стандартного типа).

Метод `CreateIoQueue` принимает следующие шесть параметров:

- ◆ `pCallbackInterface` — указатель на интерфейс `IUnknown`, который используется инфраструктурой для определения интерфейсов, реализуемых драйвером для своего объекта обратного вызова для очереди;

- ◆ *bDefaultQueue* — булево значение, указывающее, создавать ли стандартную очередь для устройства. TRUE означает стандартную очередь ввода/вывода, а FALSE — дополнительную очередь;
- ◆ *DispatchType* — одно из следующих значений перечисления `WDF_IO_QUEUE_DISPATCH_TYPE`, указывающее способ доставки запросов драйверу инфраструктурой:
  - `WdfIoQueueDispatchSequential`;
  - `WdfIoQueueDispatchParallel`;
  - `WdfIoQueueDispatchManual`;
- ◆ *bPowerManaged* — булево значение, указывающее, является ли очередь управляемой энергопотреблением;
- ◆ *bAllowZeroLengthRequests* — булево значение, указывающее, ставить ли в очередь запросы на чтение и запись, имеющие буферы нулевого размера. Значение FALSE означает, что инфраструктура должна автоматически завершить эти типы запросов для драйвера. Значение TRUE означает, что запросы этого типа направляются драйверу;
- ◆ *ppIoQueue* — указатель на переменную, принимающую указатель на интерфейс `IWDFIoQueue` для инфраструктурного объекта очереди ввода/вывода.

Образец драйвера Fx2\_Driver создает следующие три типа очередей:

- ◆ стандартную параллельную очередь для запросов на чтение и запись;
- ◆ последовательную очередь для запросов IOCTL;
- ◆ очередь с ручной диспетчеризацией, в которую драйвер помещает запросы, которые отслеживают состояние переключателей устройства.

Чтобы создать каждую из этих очередей, драйвер UMDF создает объект обратного вызова очереди, после чего вызывает инфраструктуру, чтобы создать ассоциированный инфраструктурный объект очереди. Объекты обратного вызова очереди для драйвера реализуют интерфейсы обратных вызовов, которые обрабатывают типы запросов ввода/вывода, поддерживаемые очередью.

Драйвер создает и конфигурирует очереди после создания им объекта устройства в процессе выполнения метода `OnDeviceAdd`. Все три очереди основаны на базовом классе `CMyQueue` образца драйвера. Этот класс реализует метод `Initialize`, который вызывает инфраструктуру для создания ее партнерского объекта очереди.

## Стандартная очередь UMDF

Для создания объекта обратного вызова для стандартной очереди образец драйвера Fx2\_Driver создает объект обратного вызова очереди, после чего создает партнерский инфраструктурный объект очереди, вызывая для этого метод `Initialize` базового класса. Исходный код метода `Initialize` из файла `Queue.cpp` приведен в листинге 8.3.

### Листинг 8.3. Создание и инициализация стандартной очереди UMDF

```

HRESULT CMyQueue::Initialize(
    _in WDF_IO_QUEUE_DISPATCH_TYPE DispatchType,
    _in bool Default,
    _in bool PowerManaged)
{
    IWDFIoQueue *fxQueue;
    HRESULT hr;

```

```
{ // Создается объект очереди ввода/вывода.
    IUnknown *callback = QueryIUnknown();
    hr = m_Device->GetFxDevice()->CreateIoQueue(callback,
                                                    Default,
                                                    DispatchType,
                                                    PowerManaged,
                                                    FALSE,
                                                    &fxQueue);
    callback->Release();
}
if (SUCCEEDED(hr)) {
    m_FxQueue = fxQueue;
    fxQueue->Release();
}
return hr;
}
```

Для создания инфраструктурного объекта очереди ввода/вывода метод `Initialize` вызывает метод `IWDFDevice::CreateIoQueue`. Образец драйвера `Fx2_Driver` конфигурирует стандартную очередь ввода/вывода под параллельную диспетчеризацию и управляемость энергопотреблением, а также запрещает постановку в очередь запросов с буферами нулевого размера.

Для обработки запросов ввода/вывода, для каждого объекта обратного вызова очереди драйвер реализует один или несколько интерфейсов обратных вызовов. UMDF отправляет запрос ввода/вывода драйверу, вызывая метод в соответствующем интерфейсе обратного вызова. Стандартная очередь образца драйвера поддерживает следующие интерфейсы обратного вызова UMDF:

- ◆ `IQueueCallbackRead`;
- ◆ `IQueueCallbackWrite`;
- ◆ `IRequestCallbackRequestCompletion`.

Например, чтобы отправить запрос на чтение, UMDF вызывает метод `IQueueCallbackRead::OnRead` на объекте обратного вызова очереди. Объект обратного вызова очереди может реализовывать интерфейсы, специфичные для одного типа запросов, например интерфейс `IQueueCallbackRead`, а также дополнительно реализовывать стандартный интерфейс `IQueueCallbackDefaultIoHandler`. UMDF вызывает методы интерфейса `IQueueCallbackDefaultIoHandler` для запросов на создание, чтение, запись или `IOCTL`, для которых драйвер не создал никаких других интерфейсов. Но образец драйвера `Fx2_Driver` не реализует этот интерфейс, поэтому стандартная очередь принимает только запросы на чтение и запись. Инфраструктура не исполняет другие типы запросов, для которых не имеется сконфигурированных драйвером очередей.

Стандартная очередь также реализует интерфейс `IRequestCallbackRequestCompletion`, чтобы драйвер мог предоставлять стандартную очередь в качестве объекта обратного вызова завершения для запросов, посылаемых драйвером получателю ввода/вывода.

## Нестандартные очереди UMDF

Драйвер создает нестандартные очереди почти таким же образом, как и стандартные, но с двумя отличиями:

- ◆ при вызове метода `IWDFDevice::CreateIoQueue` драйвер устанавливает значение параметра `bDefaultQueue` в `FALSE`;

- ◆ после создания драйвером инфраструктурного объекта очереди, он должен вызвать метод `ConfigureRequestDispatching`, чтобы указать типы запросов, которые инфраструктура может ставить в эту очередь.

В качестве параметров метод `ConfigureRequestDispatching` принимает значение, указывающее тип запроса, и булево значение, разрешающее или запрещающее постановку в очередь запросов указанного типа. Как `IWDFDevice`-, так и `IWDFIoQueue`-интерфейс содержат этот метод. Единственное, чем отличаются эти два метода, так это тем, что метод `IWDFDevice::ConfigureRequestDispatching` принимает дополнительный параметр, в котором предоставляется указатель на интерфейс `IWDFIoQueue` для очереди.

Образец драйвера Fx2\_Driver конфигурирует очередь для получения только запросов IOCTL. Это делается с помощью вызова метода `IWDFDevice::ConfigureRequestDispatching` следующим образом:

```
hr = m_FxDevice->ConfigureRequestDispatching  
    (m_ControlQueue->GetFxQueue(),  
     WdfRequestDeviceIoControl,  
     true);
```

Драйвер может сконфигурировать очередь для приема еще одного типа запросов, вызвав метод `ConfigureRequestDispatching` опять. Типы запросов, не указанные драйвером в вызове метода `ConfigureRequestDispatching`, направляются в стандартную очередь. Если же драйвер не создал стандартную очередь, то такие запросы либо не исполняются, либо передаются следующему нижележащему драйверу, как было описано ранее.

### **Очереди UMDF с ручной диспетчеризацией**

По мере получения драйвером запросов ввода/вывода из очереди запросов типа IOCTL, он обрабатывает их и пересыпает в очередь с ручной диспетчеризацией все запросы для извещения об изменении состояния переключателей устройства. Запросы в очередь запросов состояния переключателей ставятся только драйвером, а инфраструктура этого не делает. Драйвер завершает все эти запросы, когда пользователь переводит переключатель в другое положение. Такой запрос IOCTL предоставляет драйверу простой способ для отправки информации приложению.

Так как драйвер конфигурирует очередь для ручной диспетчеризации, для получения запросов из очереди он должен вызывать инфраструктуру. Для доставки драйверу запросов ввода/вывода из очереди с ручной диспетчеризацией применяются методы `IWDFIoQueue::RetrieveNextRequest` и `IWDFIoQueue::RetrieveNextRequestByFileObject`. Инфраструктура не вызывает драйвер для доставки запросов из очереди.

В листинге 8.4 показан код, в котором драйвер Fx2\_Driver создает эту очередь.

**Листинг 8.4. Создание очереди с ручной диспетчеризацией в драйвере UMDF**

```
    false, // Буферы нулевого размера
           // в запросе запрещены.
           &_SwitchChangeQueue);
// Освобождается ссылка, полученная интерфейсом QueryIUnknown.
SAFE_RELEASE(pQueueCallbackInterface);
```

Так как драйвер не реализует интерфейсов обратных вызовов для объекта очереди с ручной диспетчеризацией, он передает указатель на интерфейс `IUnknown` на объекте обратного вызова устройства. Для этого параметра методу `CreateIoQueue` требуется указатель, поэтому драйвер не должен передавать значение `NULL`.

## Пример KMDF: создание очередей ввода/вывода

Чтобы создать и сконфигурировать очередь ввода/вывода, драйвер KMDF выполняет такую последовательность шагов:

1. Определяет структуру `WDF_IO_QUEUE_CONFIG` для хранения установок конфигурации очереди.
2. Инициализирует эту структуру конфигурации, вызывая функцию `WDF_IO_QUEUE_CONFIG_INIT` или функцию `WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE`. В качестве параметров эти функции принимают указатель на структуру конфигурации и значение перечисления, указывающее тип диспетчеризации для очереди.
3. Устанавливает функции обратных вызовов по событию в структуре `WDF_IO_QUEUE_CONFIG` и устанавливает тип диспетчеризации: последовательный, параллельный или ручной.
4. Устанавливает требуемые булевые значения для полей `PowerManaged` и `AllowZeroLengthRequests` в структуре конфигурации очереди, если значения по умолчанию не подходят.
5. Создает очередь, вызывая метод `WdfIoQueueCreate`. Методу передаются следующие параметры: дескриптор объекта типа `WDFDEVICE`, владеющего очередью, указатель на заполненную структуру `WDF_IO_QUEUE_CONFIG`, указатель на структуру `WDF_OBJECT_ATTRIBUTES` и указатель на переменную для получения дескриптора созданной очереди.
6. Для нестандартной очереди с помощью метода `WdfDeviceConfigureRequestDispatching` указывает типы запросов ввода/вывода, которые KMDF должна направлять очереди. С помощью этого метода драйвер может также сконфигурировать стандартную очередь для получения запросов на создание.

Образец драйвера Osrusbf2 создает стандартную очередь с параллельной диспетчеризацией и две последовательные очереди следующим образом:

- ◆ для запросов IOCTL драйвер конфигурирует стандартную очередь с параллельной диспетчеризацией;
- ◆ для запросов на чтение драйвер создает очередь с последовательной диспетчеризацией;
- ◆ для запросов на запись создается еще одна очередь с последовательной диспетчеризацией.

Фрагменты кода для создания всех этих очередей были взяты из функции `OsrFxDeviceAdd` в файле `Device.c`.

## Стандартная очередь KMDF

В листинге 8.5 приведен исходный код из образца драйвера Osrusbf2 для создания стандартной очереди с параллельной диспетчеризацией для входящих запросов IOCTL.

#### Листинг 8.5. Создание стандартной очереди в драйвере KMDF

```
WDF_IO_QUEUE_CONFIG ioQueueConfig;
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQueueConfig,
                                       WdfIoQueueDispatchParallel);
ioQueueConfig.EvtIoDeviceControl = OsrFxEvtIoDeviceControl;
status = WdfIoQueueCreate(device,
                          &ioQueueConfig,
                          WDF_NO_OBJ ECT_ATTRIBUTES,
                          &queue); // Дескриптор стандартной очереди
if (!NT_SUCCESS(status)) {
    return status;
}
```

Так как создается стандартная очередь, драйвер инициализирует структуру конфигурации, вызывая функцию `WdfIoQueue_ConfigInit_Default_Queue`. В параметрах функции передается указатель на структуру конфигурации и значение `WdfIoQueueDispatchParallel` перечисления `WdfIoQueue_Dispatch_Type`, указывающее параллельную диспетчеризацию. После этого драйвер регистрирует функцию обратного вызова для событий ввода/вывода для запросов IOCTL в поле `EvtIoDeviceControl` структуры конфигурации. Так как драйвер не регистрирует никаких других функций обратного вызова для событий ввода/вывода, инфраструктура не выполняет всех других запросов ввода/вывода, для которых драйвер не сконфигурирует других очередей. Наконец, драйвер вызывает метод `WdfIoQueueCreate` для создания очереди ввода/вывода.

Если запрос отменяется, находясь в этой очереди, до того как он направлен драйверу, инфраструктура обрабатывает отмену, не извещая драйвер.

## Нестандартная очередь KMDF

Драйвер Osrusbf2 также создает отдельные очереди с последовательной диспетчеризацией для запросов на чтение и запись. Исходный код для создания и конфигурирования этих очередей показан в листинге 8.6.

#### Листинг 8.6. Создание нестандартной очереди в драйвере KMDF

```
// Создание очереди с последовательной диспетчеризацией
// для запросов на чтение и регистрация функции обратного вызова
// EvtIoStop для подтверждения незаконченных запросов
// на получателе ввода/вывода.
WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig, WdfIoQueueDispatchSequential);
ioQueueConfig.EvtIoRead = OsrFxEvtIoRead;
ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;
status = WdfIoQueueCreate(device,
                           &ioQueueConfig,
                           WDF_NO_OBJ_ECT_ATTRIBUTES,
                           &queue // Дескриптор очереди.
                     );
```

```
if (!NT_SUCCESS (status)) {
    return status;
}
status = WdfDeviceConfigureRequestDispatching(device, queue,
                                              WdfRequestTypeRead);
if (!NT_SUCCESS (status)){
    return status;
}
// Создается другая очередь с последовательной диспетчеризацией
// для запросов на запись.
WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig, WdfIoQueueDispatchSequential);
ioQueueConfig.EvtIoWrite = OsrFxEvtIoWrite;
ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;
status = WdfIoQueueCreate(device,
                         &ioQueueConfig,
                         WDF_NO_OBJ ECT_ATTRIBUTES,
                         &queue // Дескриптор очереди.
                     );
if (!NT_SUCCESS (status)) {
    return status;
}
status = WdfDeviceConfigureRequestDispatching(device, queue,
                                              WdfRequestTypeWrite);
if (!NT_SUCCESS (status)){
    return status;
}
```

Как видно из листинга 8.6, драйвер предпринимает те же самые шаги для создания как очереди на чтение, так и очереди на запись. Драйвер инициализирует очереди как нестандартные с последовательной диспетчеризацией, вызывая функцию `WDF_IO_QUEUE_CONFIG_INIT`. Первая очередь принимает только запросы на чтение, поэтому драйвер устанавливает функцию обратного вызова `EvtIoRead` в структуре `ioQueueConfig` и вызывает метод `WdfDeviceConfigureRequestDispatching`, чтобы указать инфраструктуре на необходимостьставить в очередь только запросы на чтение (передавая значение `WdfRequestTypeRead` в параметре `WDF_REQUEST_TYPE`). Вторая очередь принимает только запросы на запись, поэтому драйвер повторяет процедуру, но на этот раз, указывая значение `WdfRequestTypeWrite` для параметра `WDF_REQUEST_TYPE`.

Кроме функций обратного вызова для запросов на чтение и запись, драйвер конфигурирует обратный вызов функции `EvtIoStop` для обеих очередей, с тем, чтобы он мог должным образом обрабатывать незаконченные запросы, когда очередь останавливается. Инфраструктура вызывает функцию `EvtIoStop` для каждого запроса, принадлежащего драйверу, когда устройство выходит из рабочего состояния.

Данный функциональный драйвер конфигурирует очереди с последовательной диспетчеризацией для запросов на чтение и запись и предоставляет функцию обратного вызова `EvtIoDeviceControl` только для своей стандартной очереди. Поэтому инфраструктура завершает неудачей внутренние запросы IOCTL, возвращая статус `STATUS_INVALID_DEVICE_REQUEST`.

## Извлечение запросов из очереди с ручной диспетчеризацией

Когда драйвер готов обрабатывать запрос из очереди с ручной диспетчеризацией, он вызывает метод объекта очереди, чтобы извлечь из очереди следующий запрос.

Делается это следующим образом:

- ◆ драйверы UMDF вызывают методы интерфейса `IWDFIoQueue` объекта очереди;
- ◆ драйверы KMDF вызывают методы семейства `WdfIoQueueXxx`.

Методы, применяемые драйверами для извлечения запросов из очереди с ручной диспетчеризацией, приведены в табл. 8.10.

**Таблица 8.10. Методы для извлечения запросов из очередей с ручной диспетчеризацией**

Операция	Метод UMDF	Метод KMDF
Извлекает следующий запрос из очереди	<code>IWDFIoQueue::Retrieve Next Request</code>	<code>WdfIoQueueRetrieveNextRequest</code>
Извлекает следующий запрос для конкретного файлового объекта	<code>IWDFIoQueue:: RetrieveNextRequestByFileObject</code>	<code>WdfIoQueueRetrieveRequest ByFileObject</code>
Выполняется поиск в очереди конкретного запроса с последующим извлечением этого запроса	Никакой	<code>WdfIoQueueFindRequest</code> , а после него — метод <code>WdfIoQueueRetrieveFoundRequest</code>

По умолчанию очереди работают по принципу "первым пришел — первым обслужен". Драйвер может извлечь следующий запрос или следующий запрос для указанного объекта файла.

Драйвер может получать извещение о прибытии запроса в очередь так:

- ◆ драйвер UMDF реализует обратный вызов `IQueueCallbackStateChange` на объекте обратного вызова очереди. Когда состояние очереди меняется, инфраструктура вызывает метод `OnStateChange`, передавая ему значение, указывающее новое состояние очереди;
- ◆ драйвер KMDF вызывает метод `WdfIoQueueReadyNotify`, передавая ему указатель на функцию обратного вызова `EvtIoQueueState`. Инфраструктура вызывает функцию `EvtIoQueueState` по прибытию запроса.

С помощью метода `WdfIoQueueFindRequest` драйвер KMDF может выполнить в очереди поиск конкретного запроса. Методу передается ссылка на запрос, а он возвращает дескриптор запроса. Сам запрос из очереди не извлекается. Драйвер может проверить, является ли данный запрос тем, который ему требуется. Если нет, драйвер повторяет поиск. В случае требуемого запроса, драйвер может извлечь его из очереди с помощью метода `WdfIoQueueRetrieveFoundRequest`. После этого драйвер должен освободить ссылку на запрос, полученную методом `WdfIoQueueFindRequest`.

После того как драйвер извлечет запрос из очереди, он становится его "владельцем". Теперь драйвер должен завершить запрос, послать его другому драйверу или поставить в другую очередь.

В листинге 8.7 показан исходный код образца драйвера Pcidrv, в котором он выполняет поиск запроса с определенным кодом функции в своей очереди запросов IOCTL с ручной диспетчеризацией и извлекает этот запрос из очереди. Этот код взят из файла `Pcidrv\sys\hw\nic_req.c` и приводится с небольшими сокращениями.

**Листинг 8.7. Поиск запроса в очереди KMDF с ручной диспетчеризацией**

```
NTSTATUS NICGetIoctlRequest(IN WDFQUEUE Queue,
                             IN ULONG FunctionCode,
                             OUT WDFREQUEST* Request)
{
    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,
                                       prevTagRequest,
                                       NULL,
                                       &params,
                                       &tagRequest);
        // Метод WdfIoQueueFindRequest получает дополнительную ссылку
        // на tagRequest, чтобы предотвратить освобождение памяти.
        // Но запрос tagRequest продолжает находиться в очереди, и его
        // можно отменить или извлечь и завершить в другом потоке.
        if (prevTagRequest) {
            WdfObjectDereference(prevTagRequest);
        }
        if (status == STATUS_NO_MORE_ENTRIES) {
            status = STATUS_UNSUCCESSFUL;
            break;
        }
        if (status == STATUS_NOT_FOUND) {
            // Запрос prevTagRequest в очереди не найден.
            // Он был либо отменен, либо отправлен драйверу.
            // Могутиться другие запросы, отвечающие критериям
            // поиска, поэтому возобновляем поиск.
            prevTagRequest = tagRequest = NULL;
            continue;
        }
        if (!NT_SUCCESS(status)) {
            status = STATUS_UNSUCCESSFUL;
            break;
        }
        if (FunctionCode ==
            params.Parameters.DeviceIoControl.IoControlCode) {
            status = WdfIoQueueRetrieveFoundRequest(Queue,
                                                     tagRequest,
                                                     Request);
            WdfObjectDereference(tagRequest);
            if (status == STATUS_NOT_FOUND) {
                // Запрос prevTagRequest в очереди не найден.
                // Возобновляем поиск.
                prevTagRequest = tagRequest = NULL;
                continue;
            }
        }
    } while (status != STATUS_SUCCESS);
}
```

```

if (!NT_SUCCESS(status)) {
    status = STATUS_UNSUCCESSFUL;
    break;
}
// Запрос найден. Освобождаем дополнительную
// ссылку перед возвращением управления.
ASSERT(*Request == tagRequest);
status = STATUS_SUCCESS;
break;
}
else {
    // Запрос не тот, который требуется.
    // Освобождаем ссылку на tagRequest после
    // следующего поиска запроса.
    prevTagRequest = tagRequest;
    continue;
}
} while (TRUE);
return status;
}

```

Образец драйвера начинает поиск с вызова функции `WDF_REQUEST_PARAMETERS_INIT`, чтобы инициализировать структуру `WDF_REQUEST_PARAMETERS`. Позже, когда драйвер вызывает метод `WdfIoQueueFindRequest`, драйвер KMDF возвращает параметры для запроса в этой структуре.

Дальше драйвер инициализирует переменные, с помощью которых он отслеживает уже просмотренные запросы. Переменная `prevTagRequest` содержит дескриптор предыдущего проверенного драйвером запроса, а переменная `tagRequest` — дескриптор запроса, проверяемого в данный момент. Перед началом поиска драйвер инициализирует обе переменные значением `NULL`.

Поиск выполняется в цикле. При каждом вызове функцией `NICGetIoctlRequest` метода `WdfIoQueueFindRequest` ему передается переменная `prevTagRequest`, чтобы указать KMDF, откуда начинать поиск. Также передается указатель на переменную `tagRequest` для получения дескриптора текущего запроса. Методу `WdfIoQueueFindRequest` передается дескриптор очереди, дескриптор соответствующего объекта файла и указатель на инициализированную структуру `WDF_REQUEST_PARAMETERS`. Так как образец драйвера не использует объектов файла, для параметра дескриптора объекта файла передается значение `NULL`. Обратите внимание, что драйвер инициализирует структуру `WDF_REQUEST_PARAMETERS` повторно перед каждым вызовом, таким образом обеспечивая, что он не получит устаревшие данные.

При первом проходе цикла значение переменной `prevTagRequest` равно `NULL`. Поэтому поиск стартует в начале очереди. Метод `WdfIoQueueFindRequest` выполняет поиск в очереди и возвращает параметры запроса (в переменной `Params`) и дескриптор запроса (в переменной `tagRequest`). Чтобы предотвратить удаление запроса другим компонентом в то время, как драйвер анализирует его, метод `WdfIoQueueFindRequest` берет ссылку на запрос.

Функция `NICGetIoctlRequest` сравнивает значение кода функции, возвращаемое в структуре параметров запроса, с кодом функции, передаваемым при вызове. Если значения кодов совпадают, драйвер вызывает метод `WdfIoQueueRetrieveFoundRequest`, чтобы извлечь запрос из очереди. Методу `WdfIoQueueRetrieveFoundRequest` передаются три параметра: дескриптор объекта очереди, дескриптор, возвращенный методом `WdfIoQueueFindRequest`, указывающий запрос, который нужно извлечь из очереди, и указатель на переменную, которая получает дескриптор извлеченного запроса.

При успешном завершении метода `WdfIoQueueRetrieveFoundRequest` драйвер становится владельцем извлеченного запроса. Он удаляет полученную ранее дополнительную ссылку на запрос, вызывая для этого метод `WdfObjectDereference`. После этого драйвер выходит из цикла и функция `NICGetIoctlRequest` возвращает дескриптор извлеченного запроса. Теперь драйвер может выполнять операции ввода/вывода, требуемые для обслуживания запроса.

Если же коды функции не совпадают, драйвер присваивает переменной `prevTagRequest` значение переменной `TagRequest`, чтобы начинать следующий поиск с текущего положения в очереди. Но драйвер пока не освобождает ссылку на объект запроса, представляемый переменной `prevTagRequest`. Он должен удерживать эту ссылку до тех пор, пока метод `WdfIoQueueFindRequest` не возвратит управление в следующей итерации цикла. Потом используется следующая итерация цикла. На этот раз, если метод `WdfIoQueueFindRequest` находит запрос, драйвер удаляет ссылку, взятую методом `WdfIoQueueFindRequest` для запроса, представляемого переменной `prevTagRequest`, после чего сравнивает коды функции, как и в предыдущей итерации.

Если запроса больше нет в очереди, метод `WdfIoQueueFindRequest` возвращает статус `STATUS_NOT_FOUND`. Это может случиться, если, например, он был отменен или извлечен другим потоком. Метод `WdfIoQueueRetrieveFoundRequest` также может возвратить это же значение статуса, если дескриптор, переданный в переменной `tagRequest`, не является действительным. При любой из этих ошибок драйвер начинает поиск сначала. Если исполнение любого из этих методов завершается неуспешно по какой-либо иной причине, например, достижению конца очереди, драйвер выходит из цикла.

## Функции обратного вызова по событию

При получении запроса ввода/вывода инфраструктура может предпринять любое из следующих действий:

- ◆ поставить запрос в очередь;
- ◆ вызвать одну из функций обратного вызова драйвера, не помещая запрос в очередь;
- ◆ обработать запрос для драйвера;
- ◆ переслать запрос стандартному получателю ввода/вывода;
- ◆ завершить запрос неудачей.

Какое из предыдущих действий предпринимается инфраструктурой, зависит от типа запроса, конфигурации очередей драйвера и типа драйвера. Драйверы должны реализовывать функции обратного вызова по событию ввода/вывода и конфигурировать свои очереди таким образом, чтобы WDF могла обработать каждый тип запроса способом, соответствующим данному запросу и устройству.

## Объект файла для ввода/вывода

С точки зрения приложения пользовательского режима, весь процесс ввода/вывода выполняется через дескриптор файла. С точки зрения драйвера, файл обычно не представляет важности, если только драйверу не требуется содержать постоянную информацию о сессии. Но если к устройству могут одновременно обращаться несколько клиентов, файл предоставляет способ, с помощью которого драйвер может управлять специфичными для клиента запросами, а также выполнять специфичные для клиента действия, такие как, например, завершение всех незаконченных запросов ввода/вывода для определенного клиента.

При открытии приложением дескриптора файла менеджер ввода/вывода Windows создает файловый объект. WDF также создает инфраструктурный объект файла, который представляет файловый объект, созданный диспетчером ввода/вывода, и определяет три запроса, специфичных для файлов: на создание, на очистку и на закрытие. Драйвер может сделать выбор обрабатывать эти запросы сам или предоставить выполнение этой задачи инфраструктуре вместо него.

Драйверы UMDF должны иметь действительный файловый объект для каждого запроса ввода/вывода. Драйверы KMDF могут выполнять операции ввода/вывода, не имея соответствующего файла.

#### **Файловые объекты и запросы ввода/вывода в UMDF**

Хотя драйверы режима ядра могут посыпать и получать запросы ввода/вывода без применения файлового объекта, UMDF требует наличие действительного файлового объекта для каждого запроса ввода/вывода. Почему?

Файловый объект представляет логический сеанс, в котором происходит ввод/вывод. Требуя файловый объект для каждого запроса ввода/вывода, UMDF обеспечивает, что любой драйвер в стеке устройств UMDF может получать информацию о сеансе для запроса ввода/вывода, а драйверы могут полагать, что соответствующий файловый объект открывается для каждого запроса ввода/вывода. Кроме этого, все запросы ввода/вывода из стека устройств пользовательского режима к стеку устройств режима ядра должны проходить через Windows API, для чего требуется файл. По этой причине файловые объекты являются намного более важными для внутристекового ввода/вывода, чем они есть в KMDF. (*Правин Рао (Praveen Rao), команда разработчиков Windows Driver Foundation, Microsoft.*)

## **Автоматическая пересылка запросов на создание, очистку и закрытие**

Возможность автоматической пересылки запросов может быть полезной для драйверов, обрабатывающих одни типы запросов, а другие типы — нет. Например, драйвер фильтра UMDF может проверять данные, записываемые в файл, но не выполнять проверку запросов на создание, очистку и закрытие. Поэтому для запросов на чтение такой драйвер будет иметь интерфейс обратного вызова, но для запросов на создание, очистку и закрытие будет применять автоматическую пересылку.

Для представления извещений об очистке и закрытии инфраструктура не создает объектов запроса WDF. Поэтому драйвер может переслать такие запросы стандартному получателю ввода/вывода, только сконфигурировав автоматическую пересылку. Для запросов на создание инфраструктура создает объект запроса WDF, только если драйвер предоставит функцию обратного вызова на создание.

Какое действие инфраструктура предпринимает относительно запросов на создание, очистку и закрытие — отправляет драйверу, пересыпает или завершает — зависит от следующих обстоятельств:

- ◆ установки флага автоматической пересылки для объекта устройства;
- ◆ является ли драйвер драйвером фильтра или функциональным драйвером;
- ◆ реализует ли драйвер обратный вызов по событиям для данного типа запросов;
- ◆ для запросов на создание, сконфигурировал ли драйвер очередь для запросов.

Драйвер UMDF устанавливает автоматическую пересылку для запросов на создание, очистку и закрытие, вызывая метод `AutoForwardCreateCleanupClose` интерфейса `IWDFDeviceInitialize` перед созданием объекта устройства.

Драйвер KMDF устанавливает автоматическую пересылку для запросов на создание, очистку и закрытие, устанавливая поле `AutoForwardCleanupClose` в структуре `WDF_FILEOBJECT_CONFIG` перед созданием объекта устройства.

Драйвер указывает автоматическую пересылку с помощью константы перечисления `WDF_TRI_STATE`:

- ◆ `WdfDefault` — указывает, что инфраструктура должна руководствоваться для пересылки установками по умолчанию. Для драйверов фильтра и функциональных драйверов применяются разные установки по умолчанию, которые описываются в последующих разделах этой главы;
- ◆ `WdfTrue` — указывает, что инфраструктура должна пересылать запросы стандартному получателю ввода/вывода;
- ◆ `WdfFalse` — указывает, что инфраструктура не должна пересылать запросы.

Дополнительная информация об автоматической пересылке запросов предоставляется в последующих разделах об обработке запросов на создание, очистку и закрытие.

#### ***Дисбаланс между запросами на создание и запросами на очистку/закрытие***

Если драйвер реализует обратный вызов для запроса на создание, он должен обрабатывать все запросы на создание единообразным образом. То есть, драйвер должен или завершать все такие запросы, или отсыпал их стандартному получателю ввода/вывода. Он не может завершать одни запросы, а другие пересыпать стандартному получателю ввода/вывода.

Кроме этого, драйвер должен обрабатывать запросы на создание таким же образом, как и инфраструктура обрабатывает запросы на очистку и закрытие. Если инфраструктура пересыпает запросы на очистку и закрытие стандартному получателю ввода/вывода, драйвер также должен посыпать запросы на создание ему же. А если структура завершает запросы на очистку и закрытие, то драйвер также должен завершать запросы на создание.

Причиной этому является то, что для всех таких запросов для объекта устройства применяется флаг `AutoForwardCreateCleanupClose` в UMDF и флаг `AutoForwardCleanupClose` в KMDF. Инфраструктура не может определить, какие запросы на создание драйвер завершил, а какие переслал другому драйверу. Нижние драйверы в стеке устройств должны получать одинаковое количество запросов на создание и запросов на очистку/закрытие.

## **Обратные вызовы по событиям ввода/вывода для запросов на создание**

Драйвер должен обрабатывать запросы на создание, если его устройство может одновременно иметь несколько файлов или пользователей и если драйвер должен выполнять разные операции для каждого файла или пользователя. Обрабатывая запросы на создание, драйвер получает доступ к файловому объекту, создаваемому инфраструктурой в ответ на запрос. Файловый объект предоставляет средство, с помощью которого драйвер может различить пользователей; он также предоставляет область хранения, специфичную для пользователя, в области контекста файлового объекта.

Для обработки запросов на создание:

- ◆ для получения запросов на создание драйверы UMDF реализуют интерфейс `IQueueCallbackCreate` на объекте обратного вызова очереди или интерфейс `IQueueCallbackDefaultIoHandler` на объекте обратного вызова стандартной очереди;
- ◆ для получения запросов на создание без постановки в очередь, драйверы KMDF регистрируют обратный вызов функции `EvtDeviceFileCreate`; а для получения запросов на соз-

дание с постановкой в очередь драйвер должен зарегистрировать обратный вызов `EvtIoDefault` и сконфигурировать очередь.

Если драйвер не обрабатывает запросов на создание, WDF выполняет обработку по умолчанию, как описано в следующих разделах.

## Обработка запросов на создание в драйверах UMDF

Какое действие UMDF предпринимает относительно запроса на создание — отправляет драйверу, пересыпает, успешно завершает или завершает неудачей — зависит от того, какой тип драйвера представляет объект целевого устройства: драйвер фильтра или функциональный драйвер, а также от того, реализует ли драйвер интерфейс `IQueueCallbackCreate` на одном из своих объектов обратного вызова очереди. Варианты предпринимаемых UMDF действий приводятся в табл. 8.11.

**Таблица 8.11. Обработка запросов на создание в драйверах UMDF**

Действие драйвера	Реакция инфраструктуры
Реализует интерфейс <code>IQueueCallbackCreate</code> на объекте обратного вызова или реализует интерфейс <code>IQueueCallbackDefaultIoHandler</code> для объекта обратного вызова стандартной очереди	Ставит запросы на создание в очередь
Устанавливает <code>AutoForwardCreateCleanupClose</code> в <code>FALSE</code> (значение по умолчанию для функционального драйвера)	Открывает объект файла и завершает запрос, возвращая статус <code>S_OK</code>
Устанавливает <code>AutoForwardCreateCleanupClose</code> в <code>TRUE</code> (значение по умолчанию для драйвера фильтра)	Пересыпает запрос стандартному получателю ввода/вывода

В драйвере фильтра UMDF интерфейс `IQueueCallbackCreate` должен выполнить все требуемые задачи фильтрации, а затем либо переслать запрос стандартному получателю ввода/вывода, либо завершить запрос. Типичный драйвер фильтра перешлет запрос, чтобы нежелющие драйверы в стеке могли получить возможность завершить его. Но в случае серьезной ошибки драйвер фильтра может завершить запрос.

Для функциональных драйверов, не реализующих интерфейс `IQueueCallbackCreate`, UMDF по умолчанию открывает объект файла для представления устройства и завершает запрос, возвращая статус `S_OK`.

Пример реализации метода `OnCreateFile` на объекте обратного вызова очереди функциональным драйвером UMDF показан в листинге 8.8. Этот пример основан на коде драйвера `WpdWudfSampleDriver` из файла `Queue.cpp`.

### Листинг 8.8. Реализация метода `OnCreateFile` драйвером UMDF

```
STDMETHODIMP_ (void) CQueue::OnCreateFile(
    /*[in]*/ IWDFIoQueue*     pQueue,
    /*[in]*/ IWDFIoRequest*   pRequest,
    /*[in]*/ IWDFFile*        pFileObject
)
{
    HRESULT hr = S_OK;
    ClientContext* pClientContext = new ClientContext();
    . . . // Ненужный код пропущен.
```

```

if (pClientContext != NULL) {
    hr = pFileObject->AssignContext (this, (void*)pClientContext);
    // Если нельзя установить область контекста клиента,
    // освобождаем ее.
    if (FAILED(hr)) {
        pClientContext->Release();
        pClientContext = NULL;
    }
}
else {
    hr = E_OUTOFMEMORY;
}
pRequest->Complete(hr);
return;
}

```

Как можно видеть, метод `OnCreateFile` принимает в параметрах указатели на инфраструктурную очередь, запрос ввода/вывода и интерфейсы файла. В образце драйвера этот метод выделяет область контекста объекта для хранения специфичных для клиента данных. Чтобы ассоциировать область контекста с инфраструктурным файловым объектом, драйвер вызывает метод `IWDFObject::AssignContext`. При успешном исполнении метода `AssignContext` драйвер завершает запрос, вызывая метод `IWDFIoRequest::Complete`, передавая ему статус `S_OK`. Если драйвер не может выделить область контекста или назначить ее объекту файла, он завершает запрос, возвращая соответствующий статус неудачного исполнения.

## Имперсонация в драйверах UMDF

По умолчанию драйверы UMDF исполняются в контексте безопасности `LocalService`, который имеет минимальные привилегии безопасности на локальном компьютере и представляет параметры для анонимного сетевого доступа. Привилегии, предоставляемые этим контекстом, являются достаточными для большинства операций драйверов UMDF. Но иногда драйвер получает запрос, для выполнения которого требуются привилегии, не предоставляемые контектом `LocalService`. В такой ситуации драйвер UMDF может имперсонировать клиента и временно использовать его контекст безопасности. За исключением запросов `Plug and Play`, энергопотребления и других системных запросов, инфраструктура позволяет драйверу имперсонировать процесс клиента для обработки запросов ввода/вывода.

Максимальный уровень имперсонации, разрешаемый драйверу, указывается при установке драйвера в файле. Для этого в разделе `DDInstall.Wdf` вставляется директива `UmdfImpersonationLevel`, как показано в следующем коде:

```

[OsrUsb_Install.NT.Wdf]
...
UmdfImpersonation Level=Impersonation

```

Директива может назначить один из следующих четырех уровней имперсонации:

- ◆ `Anonymous`;
- ◆ `Identification`;
- ◆ `Impersonation`;
- ◆ `Delegation`.

По умолчанию максимальным уровнем имперсонации является `Identification`. Этот уровень позволяет драйверу использовать личность и привилегии процесса клиента, но не весь

контекст безопасности. Это такие же уровни безопасности, как и константы перечисления SECURITY\_IMPERSONATION\_LEVEL, которые именуются SecurityXxx.

В вызове метода `CreateFile` приложение может указать самый высший уровень имперсонации, который разрешается использовать драйверу. Уровень имперсонации определяет операции, которые драйвер может исполнять в контексте процесса приложения. Инфраструктура UMDF всегда использует наиболее минимальный уровень имперсонации. Если уровень имперсонации, указанный приложением, отличается от минимального уровня, указанного в INF-файле, то отражатель — который устанавливает информацию о контексте безопасности для хост-процесса драйвера — использует более низкий из этих двух уровней.

Ключом к созданию безопасных драйверов является правильный подход к работе с имперсонацией. Чтобы определить, стоит ли реализовывать имперсонацию в драйвере, необходимо знать требования безопасности к операциям, выполняемым драйвером. Например, если вы хотите оставить хороший аудиторский след, чтобы администратор мог позже определить пользователя, от чьего имени драйвер выполнил определенное действие, вам необходимо использовать уровень имперсонации `SecurityIdentification`. Но при этом следует помнить, что уровень `SecurityIdentification` может сделать доступной личную информацию, и поэтому его следует использовать лишь тогда, когда наличие аудиторского следа имеет определенную важность.

Еще один пример. Если драйверу требуется доступ к файлу в папке Мои документы пользователя, то ему будет необходимо применить уровень имперсонации `SecurityImpersonation`, т. к. пользовательский файл, скорее всего, будет иметь список ACL (Access Control List, список контроля доступа), предотвращающий доступ к нему в контексте `LocalService`.

Для получения дополнительной информации об уровнях имперсонации и безопасности пользовательского режима см. результаты поиска по `SECURITY_IMPERSONATION_LEVEL` в MSDN в Интернете по адресу <http://go.microsoft.com/fwlink/?LinkId=82952>.

Драйвер запрашивает имперсонацию, вызывая метод `IWDFIoRequest::Impersonate`, передавая в параметрах требуемый уровень имперсонации, указатель на интерфейс `IImpersonateCallback` драйвера и указатель на данные контекста. Драйвер может реализовывать интерфейс `IImpersonateCallback` на любом наиболее удобном объекте обратного вызова; обычно для этого применяется объект обратного вызова, содержащий данные контекста. Метод `Impersonate` вызывает соответствующие функции Windows API для имперсонации клиента на запрошенном уровне имперсонации, после чего вызывает функции обратного вызова драйвера `IImpersonateCallback::OnImpersonation`, передавая ей указатель на контекст. Когда функция обратного вызова `OnImpersonate` возвращает управление, метод `Impersonate` прекращает имперсонацию от имени драйвера, вызывая Windows API для возвращения к его собственной личности.

В листинге 8.9 показано, как образец драйвера `Fx2_Driver` использует имперсонацию, чтобы открыть файл с данными пользователя для записи в семиполосный индикатор. Этот исходный код находится в файле `Device.cpp`.

### Примечание

Этот код взят из обновленного примера, которого не было в текущем выпуске WDK, когда создавалась эта книга. Информацию о коде обновленного образца см. в разделе **Developing Drivers with WDF: News and Update** (Разработка драйверов с WDF: новости и обновления) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=80911>.

**Листинг 8.9. Использование имперсонации**

```
HRESULT CMYDevice::PlaybackFile(
    _in PFILE_PLAYBACK PlayInfo,
    _in IWDFIoRequest *FxRequest)
{
    PLAYBACK_IMPERSONATION_CONTEXT context = {PlayInfo, NULL, S_OK};
    . . . // Дополнительные объявления опущены.
    HRESULT hr;
    // Выполняем имперсонацию и открываем файл для воспроизведения.
    hr = FxRequest->Impersonate(SecurityImpersonation,
                                  this->QueryIImpersonateCallback(),
                                  &context);
    this->Release();
    hr = context.Hr;
    if (FAILED(hr)) {
        goto exit;
    }
    . . . // Код для чтения файла и кодирования данных
    // для 7-полосного индикатора опущен.
    return hr;
}
```

Драйвер вызывает метод `PlaybackFile` в листинге 8.9 в ответ на IOCTL-запрос `IOCTL_OSRUSBFX2_PLAY_FILE`. Буфер ввода для этого запроса содержит имя файла. После проверки драйвером имени файла, он вызывает метод `PlaybackFile`, чтобы открыть файл, считать из него данные и записать их на индикатор устройства.

Метод `PlaybackFile` инициализирует данные для метода `OnImpersonate`. Следующие три поля представляют информацию, требуемую драйвером, чтобы открыть файл и возвратить дескриптор и статус:

- ◆ `PlayInfo` — имя файла, который нужно открыть; инициализируется значением `PlayInfo`;
- ◆ `FileHandle` — дескриптор открытого файла; инициализируется значением `NULL`;
- ◆ `Hr` — результат операции; инициализируется значением `S_OK`.

После этого метод `PlaybackFile` вызывает метод `IWDFIoRequest::Impersonate`, передавая в параметрах значение `SecurityImpersonation` для требуемого уровня имперсонации, указатель на интерфейс `IImpersonateCallback` и указатель на данные контекста.

UMDF выполняет имперсонацию клиента на уровне `SecurityImpersonation`, после чего вызывает метод драйвера `OnImpersonate`. Метод `OnImpersonate` должен выполнить только те задачи, для которых требуется имперсонация, и не должен вызывать никаких других инфраструктурных методов. Обычно метод `OnImpersonate` выполняет одну простую задачу, такую как, например, открытие файла, защищенного списком ACL. После того как файл открыт, драйвер может выполнять с ним операции чтения и записи, не требуя имперсонации клиента.

В листинге 8.10 показаны определения метода `OnImpersonate`, который вызывается в листинге 8.9. Исходный код этого примера также находится в файле `Device.cpp`.

Как можно видеть в листинге, метод `OnImpersonate` вызывает функцию Windows `CreateFile`, чтобы открыть файл, и больше ничего не делает. Метод возвращает в области контекста дескриптор файла и статус.

**Листинг 8.10. Функция обратного вызова OnImpersonate**

```

VOID CMYDevice::OnImpersonate(_in PVOID Context)
{
    PPLAYBACK_IMPERSONATION_CONTEXT context;
    context = (PPLAYBACK_IMPERSONATION_CONTEXT) Context;
    context->FileHandle = CreateFile(context->PlaybackInfo->Path,
                                      CENERIC_READ,
                                      FILE_SHARE_READ,
                                      NULL,
                                      OPEN_EXISTING,
                                      FILE_ATTRIBUTE_NORMAL,
                                      NULL);

    if (context->FileHandle == INVALID_HANDLE_VALUE) {
        DWORD error = GetLastError();
        context->Hr = HRESULT_FROM_WIN32(error);
    }
    else {
        context->Hr = S_OK;
    }
    return;
}

```

**Обработка запросов на создание в драйверах KMDF**

Инфраструктура KMDF может отправлять запросы на создание драйверу или пересылать или завершать их. Конкретное предпринимаемое действие зависит от того, представляет ли объект устройства драйвер фильтра, зарегистрировал ли драйвер функцию обратного вызова *EvtDeviceFileCreate* для файлового объекта и сконфигурировал ли драйвер очередь для получения запросов на создание. Возможные варианты реакции KMDF на запросы на создание приведены в табл. 8.12.

**Таблица 8.12. Обработка запросов на создание в драйверах KMDF**

Действие драйвера	Реакция инфраструктуры KMDF
Конфигурирует очередь для <i>WdfRequestTypeCreate</i> и регистрирует обратный вызов функции <i>EvtIoDefault</i> для этой очереди	Ставит запросы на создание в очередь
Регистрирует функцию обратного вызова <i>EvtDeviceFileCreate</i> , вызывая метод <i>WdfDeviceInitSetFileObjectConfig</i> из функции <i>EvtDriverDeviceAdd</i>	Активирует функцию обратного вызова <i>EvtDeviceFileCreate</i> и не ставит запросы на создание в очередь
Драйвер является функциональным драйвером и ни конфигурирует очередь для получения запросов на создание, ни регистрирует обратный вызов функции <i>EvtDeviceFileCreate</i>	Открывает объект файла и завершает запросы на создание, возвращая статус <i>STATUS_SUCCESS</i>
Драйвер является драйвером фильтра и ни конфигурирует очередь для получения запросов на создание, ни регистрирует обратный вызов функции <i>EvtDeviceFileCreate</i>	Пересыпает запросы на создание стандартному получателю ввода/вывода

Инфраструктура не добавляет запросы на создание в стандартную очередь автоматически. Драйвер должен вызвать метод `WdfDeviceConfigureRequestDispatching`, чтобы явным образом сконфигурировать стандартную очередь для приема запросов на создание и должен реализовать обратный вызов функции `EvtIoDefault` для очереди.

Если драйвер KMDF не обрабатывает запросы на создание, KMDF по умолчанию пересыпает все запросы на создание, очистку и закрытие стандартному получателю ввода/вывода. Драйверы фильтра, обрабатывающие запросы на создание, должны выполнить все требуемые задачи фильтрации, после чего переслать запрос стандартному получателю ввода/вывода.

Если вместо пересылки запросов на создание драйвера фильтра завершает их, он должен присвоить значение `WdfFalse` члену перечисления `AutoForwardCleanupClose` структуры конфигурации, чтобы KMDF завершала запросы на очистку и закрытие для объекта файла, вместо того, чтобы пересыпать их.

Если функциональный драйвер или драйвер шины KMDF не регистрирует обратный вызов, ни конфигурирует очередь для запросов на создание, то KMDF обрабатывает эти запросы для драйвера, завершая запрос с возвратом статуса `STATUS_SUCCESS`. Но это стандартное поведение имеет один нежелательный побочный эффект. Даже если драйвер не регистрирует интерфейс устройства, вредоносное приложение может попытаться открыть устройство, используя имя его объекта PDO, и инфраструктура успешно выполнит запрос. В результате драйверы, которые не поддерживают запросов на создание, должны зарегистрировать обратный вызов функции `EvtDeviceFileCreate`, которая явным образом завершает такие запросы неудачей. Это поведение отличается от поведения в WDM, где такие запросы завершаются неудачей по умолчанию.

## **Внимание!**

Функциональные драйверы и драйверы шины, которые не принимают запросов на создание или открытие из приложений пользовательского режима — и поэтому не регистрируют интерфейса устройства — должны зарегистрировать обратный вызов функции `EvtDeviceFileCreate`, которая явным образом завершает такие запросы неудачей. Организация обратного вызова для завершения запросов неудачей обеспечивает, что зловредным приложениям перекрывается доступ к устройству.

В листинге 8.11 показан пример регистрации обратного вызова функции *EvtDeviceFileCreate* образцом драйвера Featured Toaster. Этот код содержится в функции обратного вызова *EvtDriverDeviceAdd* образца драйвера (т. е. *ToasterEvtDeviceAdd*) в файле *Toaster.c*.

#### Листинг 8.11. Регистрация функции обратного вызова EvtDeviceFileCreate

Драйвер регистрирует свои обратные вызовы перед созданием объекта устройства. Как можно видеть из листинга, образец драйвера регистрирует обратные вызовы для запросов на создание и закрытие, вызывая функцию `WDF_FILEOBJECT_CONFIG_INIT`. Эта функция принимает указатель на выделенную драйвером структуру `WDF_FILEOBJECT_CONFIG` в первом параметре, после которого идут указатели на три возможные функции обратного вызова объекта файла: `EvtDeviceFileCreate`, `EvtFileClose` и `EvtFileCleanup`. После этого драйвер вызывает метод `WdfDeviceInitSetFileObjectConfig`, чтобы инициализировать эти установки в структуре `WDFDEVICE_INIT`, которая была передана функции обратного вызова `EvtDriverDeviceAdd`.

Если функция обратного вызова `EvtDeviceFileCreate` отправляет запрос на создание стандартному получателю ввода/вывода, функция обратного вызова завершения ввода/вывода не должна менять статус завершения запроса на создание. Причиной этому требованию является отсутствие способа для нижележащих драйверов в стеке узнать об изменении статуса. Если драйвер KMDF изменит статус неуспеха на успех, нижние драйверы не смогут обрабатывать последующие запросы ввода/вывода. И наоборот, если драйвер KMDF изменит статус успеха на неуспех, нижние драйверы не смогут определить, что дескриптор файла не был создан и никогда не получат запросов на очистку или закрытие для объекта файла.

## Обратные вызовы по событиям ввода/вывода для очистки и закрытия

WDF может вызвать драйвер для обработки извещений об очистке и закрытии или может автоматически направить их стандартному получателю ввода/вывода. Ни KMDF, ни UMDF не ставят запросы на очистку или закрытие в очередь.

Для обработки запросов на очистку и закрытие:

- ◆ драйверы UMDF реализуют интерфейсы `IFileCallbackCleanup` и `IFileCallbackClose` на объекте обратного вызова устройства;
- ◆ драйверы KMDF регистрируют функции обратного вызова `EvtFileCleanup` и `EvtFileClose`.

В зависимости от его специфичных требований для операций очистки и закрытия, драйвер может реализовывать поддержку вызова одной, обеих или никакой из этих функций.

Действия, предпринимаемые WDF по прибытию запроса на очистку или закрытие, приведены в табл. 8.13.

**Таблица 8.13. Обработка запросов на очистку и закрытие инфраструктурой WDF**

Действие драйвера UMDF	Действие драйвера KMDF	Реакция WDF
Реализует интерфейс <code>IFileCallbackCleanup</code> или <code>IFileCallbackClose</code>	Регистрирует функцию обратного вызова <code>EvtFileCleanup</code> или <code>EvtFileClose</code> во время исполнения функции <code>EvtDriverDeviceAdd</code>	Активирует обратный вызов для запроса очистки или закрытия
Не реализует обратного вызова для очистки или закрытия и является функциональным драйвером — Или — Устанавливает значение <code>AutoForwardCreateCleanupClose</code> в <code>WdfFalse</code> , не реализует соответствующей функции обратного вызова и является драйвером фильтра	Не регистрирует обратного вызова для очистки или закрытия и является функциональным драйвером — Или — Устанавливает значение члена <code>AutoForwardCleanupClose</code> структуры <code>FILE_OBJECT_CONFIG</code> в <code>WdfFalse</code> , не реализует соответствующей функции обратного вызова и является драйвером фильтра	Завершает запрос, возвращая статус <code>S_OK</code> . (Только в UMDF.) — Или — Завершает запрос, возвращая статус <code>STATUS_SUCCESS</code> . (Только в KMDF.)

Таблица 8.13 (окончание)

Действие драйвера UMDF	Действие драйвера KMDF	Реакция WDF
Не реализует обратного вызова для очистки или закрытия и является драйвером фильтра. — Или — Устанавливает значение AutoForwardCreateCleanupClose в WdfTrue и является функциональным драйвером	Не регистрирует обратного вызова для очистки или закрытия и является драйвером фильтра. — Или — Устанавливает значение AutoForwardCleanupClose в WdfTrue и является функциональным драйвером	Пересыпает запросы на очистку и закрытие стандартному получателю ввода/вывода

Инфраструктура вызывает функцию обратного вызова драйвера для очистки файла по закрытию и освобождению последнего дескриптора файлового объекта, когда файл больше не имеет клиентов. Обратный вызов для зачистки должен отменить все незаконченные запросы ввода/вывода для файла.

Инфраструктура вызывает функцию обратного вызова для закрытия файла после возвращения управления функцией очистки и после завершения всех незаконченных запросов ввода/вывода для файла. При исполнении функции обратного вызова очистки устройство может не находиться в рабочем состоянии. Функция обратного вызова для закрытия файла должна отменить все выделения ресурсов, выполненные драйверов для файла.

Для драйверов KMDF инфраструктура одновременно вызывает функцию *EvtFileClose* в контексте произвольного потока.

Регистрация драйвером KMDF функций обратного вызова для создания и закрытия файла показана в листинге 8.11.

## Функции обратного вызова для запросов на чтение, запись и IOCTL

Для запросов на чтение, запись, IOCTL и внутренних IOCTL драйвер создает одну или несколько очередей и конфигурирует каждую очередь для получения одного или нескольких типов запросов ввода/вывода. На рис. 8.5 показана блок-схема алгоритма, по которому инфраструктура определяет, какое действие предпринять с запросами этих типов.

Как можно видеть на рис. 8.5, по прибытию запроса на чтение, запись, IOCTL или внутреннего IOCTL выполняются следующие действия:

1. Инфраструктура определяет, сконфигурировал ли драйвер очередь для данного типа запросов. Если нет, то для функционального драйвера и для драйвера шины инфраструктура завершает запрос неудачей. Для драйвера фильтра инфраструктура передает запрос стандартному получателю ввода/вывода.
2. Инфраструктура определяет, принимает ли данная очередь запросы. Если нет, то инфраструктура завершает запрос неуспехом.
3. Если очередь принимает запросы, то инфраструктура создает объект запроса WDF, представляющий запрос, и додает его в очередь.
4. Если устройство не находится в рабочем состоянии, то инфраструктура извещает обработчика Plug and Play и энергопотребления о необходимости перевести устройство в рабочее состояние (D0).

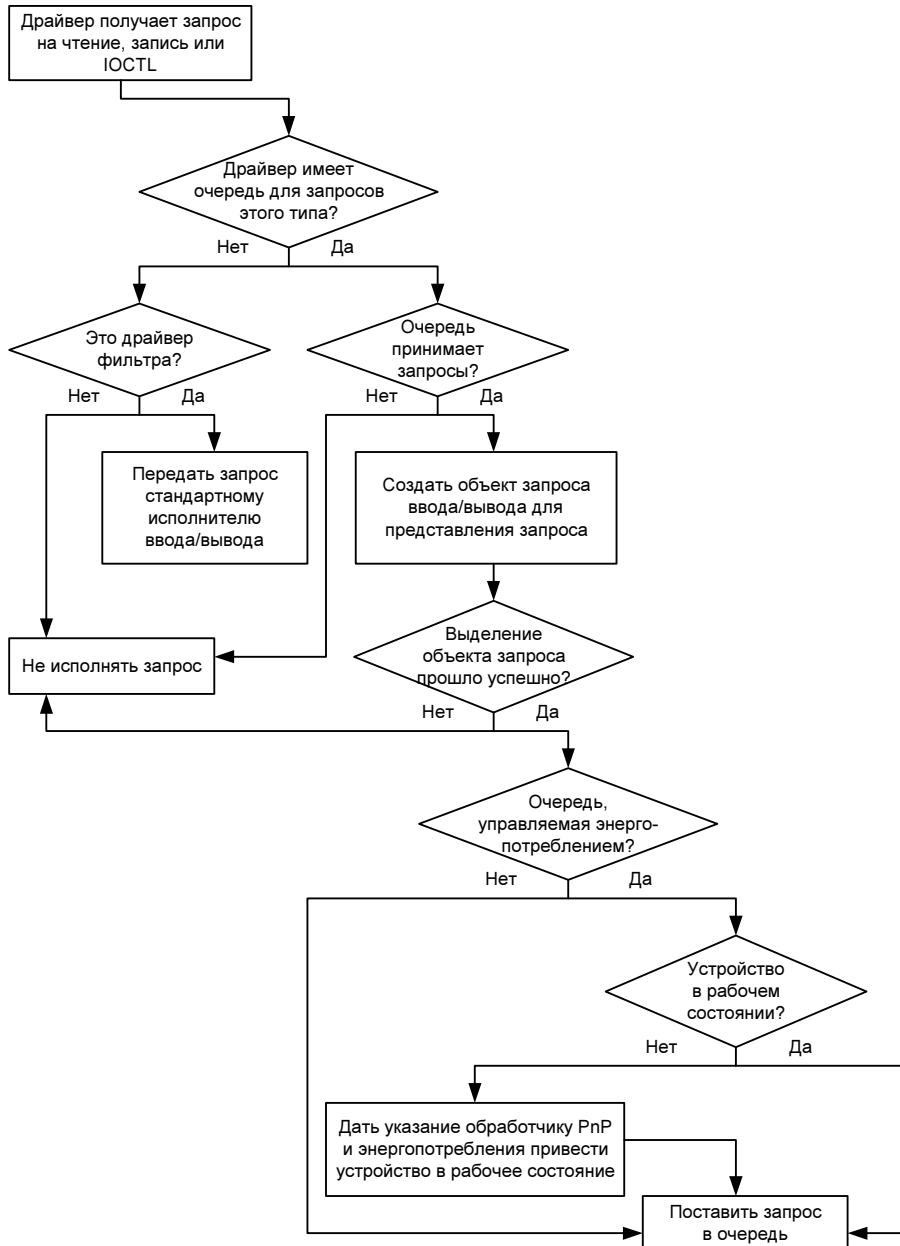


Рис. 8.5. Поток запросов ввода/вывода через инфраструктуру

В следующих разделах представлены дополнительные подробности и примеры функций обратного вызова для запросов на чтение, запись и IOCTL.

### Обратные вызовы для запросов на чтение и запись

Драйверы обрабатывают запросы на чтение и запись с помощью обратных вызовов функций, приведенных в табл. 8.14.

**Таблица 8.14. Обратные вызовы для запросов на чтение и запись**

Тип запроса	Интерфейсы обратного вызова очереди UMDF	Функции обратного вызова по событию KMDF
На чтение	IQueueCallbackRead ИЛИ IQueueCallbackDefaultIo Handler	EvtIoRead ИЛИ EvtIoDefault
На запись	IQueueCallbackWrite ИЛИ IQueueCallbackDefaultIo Handler	EvtIoWrite ИЛИ EvtIoDefault

Функциям обратного вызова для чтения и записи при вызове передается следующая информация:

- ◆ очередь, отправляющая запрос, выраженная указателем на интерфейс `IWDFQueue` очереди или дескриптором объекта типа `WDFQUEUE`;
  - ◆ сам запрос, выраженный указателем на интерфейс `IWDFIoRequest` объекта запроса или дескриптором объекта типа `WDFREQUEST`;
  - ◆ количество байтов, которые нужно считать или записать.

В функции обратного вызова драйвер может вызывать методы объекта запроса, чтобы извлечь дополнительную информацию о запросе, такую как, например, указатели на буферы ввода и вывода. После извлечения драйвером требуемой им информации он может начать выполнение операции ввода/вывода.

Для получения дополнительной информации об извлечении параметров см. разд. "Буферы ввода/вывода и объекты памяти" ранее в этой главе.

### Пример UMDF: метод *IQueueCallbackWrite::OnWrite*

В листинге 8.12 приводится метод `IQueueCallbackWrite::OnWrite`, который образец драйвера `Fx2_Driver` реализует для своего объекта обратного вызова очереди для чтения и записи. Исходный код этого метода находится в файле `ReadWriteQueue.cpp`.

**Листинг 8.12. Метод `IQueueCallbackWrite::OnWrite` в драйвере UMDF**

```

if (FAILED(hr)) {
    pWdfRequest->Complete(hr);
}
else {
    ForwardFormattedRequest(pWdfRequest, pOutputPipe);
}
SAFE_RELEASE(pInputMemory);
return;
}

```

При вызове методу `onWrite` передаются три параметра: указатель на интерфейс `IWDFIoQueue` для инфраструктурного объекта очереди, указатель на интерфейс `IWDFIoRequest` для инфраструктурного объекта запроса и количество байтов, которые нужно записать.

Драйвер принимает запрос на чтение от клиента и отсылает его каналу USB получателю ввода/вывода, который выполняет ввод/вывод для устройства. В результате этот метод сначала инициализирует некоторые переменные, после чего вызывает метод `IWDFIoRequest::GetInputMemory`, чтобы получить указатель на интерфейс `IWDFMemory` объекта памяти, который встроен в запрос. Прежде чем драйвер может послать запрос получателю ввода/вывода канала USB, он должен вызвать метод `IWDFIoTarget::FormatRequestForWrite` получателя ввода/вывода, чтобы отформатировать запрос должным образом. Для этого метода требуется указатель на интерфейс `IWDFMemory`. В случае успешного форматирования запроса драйвер пересыпает его получателю ввода/вывода и возвращает управление.

Дополнительную информацию об использовании получателей ввода/вывода USB см. в главе 9.

### Пример KMDF: функция обратного вызова `EvtIoRead`

Образец драйвера `Osrusbf2` конфигурирует функцию обратного вызова по событию `EvtIoRead` для одной из своих очередей с последовательной диспетчеризацией следующим образом:

```

WDF_IO_QUEUE_CONFIG_INIT(&ioQueueConfig, WdfIoQueueDispatchSequential);
ioQueueConfig.EvtIoRead = OsrFxEvtIoRead;
ioQueueConfig.EvtIoStop = OsrFxEvtIoStop;
status = WdfIoQueueCreate(device,
                           &ioQueueConfig,
                           WDF_NO_OBJ_ECT_ATTRIBUTES,
                           &queue);

```

Когда для объекта устройства прибывает запрос на чтение, инфраструктура становится в очередь с последовательной диспетчеризацией. По завершению драйвером обработки предыдущего запроса из этой очереди, инфраструктура активирует обратный вызов функции `EvtIoRead` для отправки запроса драйверу. Исходный код этой функции показан в листинге 8.13.

#### Листинг 8.13. Функция обратного вызова `EvtIoRead` в драйвере KMDF

```

VOID OsrFxEvtIoRead(IN WDFQUEUE Queue,
                     IN WDFREQUEST Request,
                     IN size_t Length)
{
    WDFUSBPIPE pipe;
    NTSTATUS status;
}

```

```
WDFMEMORY           reqMemory;
PDEVICE_CONTEXT     pDeviceContext;
UNREFERENCED_PARAMETER(Queue);
// Сначала проверяем достоверность параметров ввода.
if (Length > TEST_BOARD_TRANSFER_BUFFER_SIZE) {
    status = STATUS_INVALID_PARAMETER;
    goto Exit;
}
pDeviceContext = GetDeviceContext(WdfIoQueueGetDevice(Queue));
pipe = pDeviceContext->BulkReadPipe;
status = WdfRequestRetrieveOutputMemory(Request, &reqMemory);
if (!NT_SUCCESS(status)) {
    goto Exit;
}
// В вызываемом методе форматирования выполняется проверка
// достоверности типа канала, устанавливаются флаги передачи,
// создается пакет URB (USB Request Block, блок запроса USB)
// и инициализируется запрос.
status = WdfUsbTargetPipeFormatRequestForRead(pipe,
                                               Request,
                                               reqMemory,
                                               NULL //Offsets
                                              );
if (!NT_SUCCESS(status)) {
    goto Exit;
}
WdfRequestSetCompletionRoutine(Request,
                               EvtRequestReadCompletionRoutine,
                               pipe);
// Посыпаем запрос асинхронно.
if (WdfRequestSend(Request, WdfUsbTargetPipeGetIoTarget(pipe),
                    WDF_NO_SEND_OPTIONS) == FALSE) {
    // Инфраструктура не может послать запрос по какой-то причине.
    status = WdfRequestGetStatus(Request);
    goto Exit;
}
Exit:
if (!NT_SUCCESS(status)) {
    WdfRequestCompleteWithInformation(Request, status, 0);
}
return;
```

Как показано в листинге, при вызове функции *EvtIoRead* ей передается дескриптор очереди ввода/вывода, дескриптор объекта запроса и значение, указывающее количество байтов, которые нужно прочитать. Драйвер сначала проверяет достоверность значения количества байтов для чтения, после чего извлекает информацию из области контекста устройства. Драйвер извлекает буфер, в который он будет считывать данные, вызывая метод *WdfRequestRetrieveOutputMemory*, после чего форматирует запрос для канала USB получателя ввода/вывода. После установки обратного вызова для завершения ввода/вывода драйвер посыпает запрос каналу получателю. В случае ошибки драйвер завершает запрос, возвращая статус неудачи.

Дополнительную информацию об извлечении буферов из запросов ввода/вывода см. в разд. "Извлечение буферов в драйверах KMDF" ранее в этой главе. Для получения дополнительной информации об отправке запросов и использовании получателей ввода/вывода см. главу 9.

## Обратные вызовы для запросов IOCTL

Драйверы обрабатывают запросы IOCTL с помощью обратных вызовов функций, приведенных в табл. 8.15.

**Таблица 8.15. Обратные вызовы для запросов IOCTL**

Тип запроса	Интерфейсы обратного вызова UMDF	Функции обратного вызова по событию KMDF
IOCTL	IQueueCallbackDeviceIoControl или IQueueCallbackDefaultIoHandler	EvtIoDeviceControl или EvtIoDefault
Внутренний IOCTL	Никакой	EvtIoInternalDeviceControl или EvtIoDefault

Функциям обратного вызова для запросов IOCTL передается следующая информация.

- ◆ Очередь, отправляющая запрос. Выражается указателем на интерфейс IWDFQueue объекта очереди или дескриптором объекта WDFQUEUE.
- ◆ Сам запрос. Выражается указателем на интерфейс IWDFIoRequest объекта запроса или дескриптором объекта WDFREQUEST.
- ◆ Код IOCTL. Выражается значением ULONG.
- ◆ Размер буфера ввода. Выражается значением SIZE\_T.
- ◆ Размер буфера вывода. Выражается значением SIZE\_T.

### Пример UMDF: метод *OnDeviceIoControl*

В листинге 8.14 приведен код метода IQueueCallbackDeviceIoControl::OnDeviceIoControl, который образец драйвера Fx2\_Driver реализует на своем объекте обратного вызова очереди запросов IOCTL. Эта функция находится в исходном файле ControlQueue.cpp. В целях облегчения понимания кода здесь показан метод из версии драйвера Step3, а не из версии Final.

#### Листинг 8.14. Метод IQueueCallbackDeviceIoControl::OnDeviceIoControl в драйвере UMDF

```
VOID STDMETHODCALLTYPE CMyControlQueue::OnDeviceIoControl(
    _in IWDFIoQueue *FxQueue,
    _in IWDFIoRequest *FxRequest,
    _in ULONG ControlCode,
    _in SIZE_T InputBufferSizeInBytes,
    _in SIZE_T OutputBufferSizeInBytes)
{
    UNREFERENCED_PARAMETER(FxQueue);
    UNREFERENCED_PARAMETER(OutputBufferSizeInBytes);
    IWDFMemory *memory = NULL;
    PVOID buffer;
    SIZE_T bigBufferCb;
    ULONG information = 0;
```

```
bool completeRequest = true;
HRESULT hr = S_OK;
switch (ControlCode)
{
    case IOCTL_OSRUSBFX2_SET_BAR_CRAPH_DISPLAY: {
        // Проверяем, что буфер достаточного размера.
        if (InputBufferSizeInBytes < sizeof(BAR_CRAPH_STATE)) {
            hr = HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER);
        }
        else {
            FxRequest->GetInputMemory(&memory);
        }
        // Извлекаем буфер данных и используем его для
        // организации столбчатой диаграммы.
        if (SUCCEEDED(hr)) {
            buffer = memory->GetDataBuffer(&bigBufferCb);
            memory->Release();
            hr=m_Device->SetBarCraphDisplay((PBAR_CRAPH_STATE) buffer);
        }
        break;
    }
    default: {
        hr = HRESULT_FROM_WIN32(ERROR_INVALID_FUNCTION);
        break;
    }
}
if (completeRequest) {
    FxRequest->CompleteWithInformation(hr, information);
}
return;
}
```

После инициализации нескольких локальных переменных метод анализирует значение кода управления, чтобы определить, какое действие предпринять. Эта версия драйвера обрабатывает только один код управления, который устанавливает светодиодную линейку. Если запрос содержит этот код, то драйвер сверяет размер буфера с размером, поддерживаемым устройством. Если буфер достаточного размера, то драйвер извлекает объект памяти, вызывая метод `IWDFMemory::GetInputMemory` объекта памяти, после чего извлекает из этого объекта памяти буфер, вызывая метод `IWDFMemory::GetDataBuffer`. Затем драйвер устанавливает светодиодный индикатор и завершает запрос.

Для получения дополнительной информации об извлечении объектов памяти и буферов данных из запросов ввода/вывода и завершении запросов ввода/вывода см. разд. "Извлечение буферов в драйверах UMDF" ранее в этой главе.

### Пример KMDF: функция обратного вызова `EvtIoDeviceControl`

В листинге 8.15 показано, как драйвер KMDF обрабатывает запрос IOCTL. Этот пример взят из файла Toaster.c для образца драйвера Featured Toaster.

#### Листинг 8.15. Функция обратного вызова `EvtIoDeviceControl` в драйвере KMDF

```
VOID ToasterEvtIoDeviceControl(
    IN WDFQUEUE Queue,
```

```

    IN WDFREQUEST Request,
    IN size_t OutputBufferLength,
    IN size_t InputBufferLength,
    IN ULONG IoControlCode)
{
    NTSTATUS status= STATUS_SUCCESS;
    WDF_DEVICE_STATE deviceState;
    WDFDEVICE hDevice = WdfIoQueueGetDevice(Queue);
    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(InputBufferLength);
    PACED_CODE();
    switch (IoControlCode) {
        case IOCTL_TOASTER_DONT_DISPLAY_IN_UI_DEVICE:
            // При подгонке этого образца под конкретное аппаратное
            // обеспечение этот код необходимо удалить.
            WDF_DEVICE_STATE_INIT(&deviceState);
            deviceState.DontDisplayInUI = WdfTrue;
            WdfDeviceSetDeviceState(hDevice, &deviceState);
            break;
        default:
            status = STATUS_INVALID_DEVICE_REQUEST;
    }
    // Завершаем запрос.
    WdfRequestCompleteWithInformation(Request, status, (ULONG_PTR) 0);
}

```

Образец драйвера Toaster обрабатывает только один код IOCTL, который отключает отображение тостера в менеджере устройств. Когда прибывает запрос IOCTL для драйвера, инфраструктура ставит запрос в стандартную очередь драйвера и активирует обратный вызов функции *EvtIoDeviceControl*, который драйвер сконфигурировал для очереди.

Функция обратного вызова *ToasterEvtIoDeviceControl* анализирует значение, переданное инфраструктурой в параметре *IoControlCode*. Если значение кода равно *IOCTL\_TOASTER\_DONT\_DISPLAY\_IN\_UI\_DEVICE*, то драйвер инициализирует структуру *WDF\_DEVICE\_STATE*, устанавливает значение члена *DontDisplayInUI* в *WdfTrue* и вызывает метод *WdfDeviceSetDeviceState* для установки нового состояния. Если параметр *IoControlCode* содержит любое другое значение, то драйвер устанавливает статус неуспеха. После этого драйвер завершает запрос.

## Стандартные обратные вызовы для ввода/вывода

Как KMDF-, так и UMDF-драйверы могут реализовывать стандартные обратные вызовы для ввода/вывода, которые активируются для типов запросов ввода/вывода, для которых драйвер сконфигурировал очередь, но не предоставил никаких других обратных вызовов. Стандартные обратные вызовы для ввода/вывода получают только запросы на чтение, запись, IOCTL и внутренний IOCTL; они не могут обрабатывать запросы на создание.

Стандартные обратные вызовы для ввода/вывода получают следующую информацию.

- ◆ **Очередь, отправляющая запрос.** Выражается указателем на интерфейс *IWDFIoQueue* объекта очереди или дескриптором объекта *WDFQUEUE*.
- ◆ **Запроса ввода/вывода.** Выражается указателем на интерфейс *IWDFIoRequest* объекта запроса или дескриптором объекта *WDFREQUEST*.

Стандартный обратный вызов должен извлечь параметры из объекта запроса, чтобы определить тип запроса и, соответственно, способ его обработки.

### Пример UMDF: интерфейс *IQueueCallbackDefaultIoHandler*

В листинге 8.16 показан стандартный обработчик ввода/вывода для образца драйвера Usb\Filter. Драйвер создает одну очередь и организовывает два обратных вызова на объекте обратного вызова очереди: *IQueueCallbackWrite* для запросов на запись и *IQueueCallbackDefaultIoHandler* для всех других запросов. Драйвер фильтрует запросы на запись по их прибытию, а запросы на чтение после их завершения нижними драйверами. Драйвер реализует интерфейс стандартного обработчика ввода/вывода, чтобы он мог организовать обратный вызов для завершения запросов на чтение.

#### Листинг 8.16. Интерфейс *IQueueCallbackDefaultIoHandler::OnDefaultIoHandler* в драйвере UMDF

```
void CMyQueue::OnDefaultIoHandler(
    in IWDFIoQueue* FxQueue,
    in IWDFIoRequest* FxRequest)
{
    UNREFERENCED_PARAMETER(FxQueue);
    // Пересылаем запрос вниз по стеку. По завершению обработки
    // нижним устройством будет вызвана OnComplete,
    // чтобы завершить запрос.
    ForwardRequest(FxRequest);
}
```

Метод *IQueueCallbackDefaultIoHandler::OnDefaultIoHandler* в данном листинге не представляет собой ничего сложного. Он не выполняет никакой обработки запроса, а просто вызывает вспомогательную функцию, чтобы переслать запрос стандартному получателю ввода/вывода, после чего передает управление обратно. Но вспомогательная функция организовывает обратный вызов для завершения ввода/вывода, т. к. драйвер фильтрует запросы на чтение после их завершения нижележащими драйверами.

Дополнительную информацию об обратных вызовах для завершения ввода/вывода см. в главе 9.

### Пример KMDF: функция обратного вызова *EvtIoDefault*

В листинге 8.17 показана функция обратного вызова *EvtIoDefault* из образца драйвера AMCC5933. Исходный код этой функции содержится в файле AMCC5933\Sys\Transfer.c.

#### Листинг 8.17. Функция обратного вызова *EvtIoDefault* в драйвере KMDF

```
VOID AmccPciEvtIoDefault(IN WDFQUEUE Queue,
                           IN WDFREQUEST Request)
{
    PAMCC_DEVICE_EXTENSION devExt;
    REQUEST_CONTEXT * transfer;
    NTSTATUS status;
    size_t length;
    WDF_DMA_DIRECTION direction;
    WDFDMA TRANSACTION dmaTransaction;
```

```

WDF_REQUEST_PARAMETERS      params;
WDF_REQUEST_PARAMETERS_INIT(&params);
WdfRequestCetParameters(Request, &params);
// Получаем область контекста устройства.
devExt = AmccPciCetDevExt(WdfIoQueueCetDevice(Queue));
// Проверяем достоверность параметров и сохраняем их.
switch (params.Type) {
    case WdfRequestTypeRead:
        length      = params.Parameters.Read.Length;
        direction   = WdfDmaDirectionReadFromDevice;
        break;
    case WdfRequestTypeWrite:
        length      = params.Parameters.Write.Length;
        direction   = WdfDmaDirectionWriteToDevice;
        break;
    default:
        WdfRequestComplete(Request, STATUS_INVALID_PARAMETER);
        return;
}
// Размер должен быть ненулевым.
if (length == 0) {
    WdfRequestComplete(Request, STATUS_INVALID_PARAMETER);
    return;
}
// Код продолжает организовывать операцию DMA.
...
return;
}

```

Функция обратного вызова *EvtIoDefault* принимает только два параметра: дескриптор объекта очереди и дескриптор объекта запроса ввода/вывода. В драйвере AMCC5933 функция *EvtIoDefault* обрабатывает запросы на чтение, запись и IOCTL, поэтому первым делом она получает параметры запроса, вызывая для этого метод *WdfGetRequestParameters*. Этот метод возвращает указатель на структуру *WDF\_REQUEST\_PARAMETERS*. Поле *Type* этой структуры содержит идентификатор типа запроса, который, в свою очередь, определяет направление операции ввода/вывода. Если запрос не является запросом на чтение или запись либо если запрошена передача нулевого количества байтов, драйвер завершает запрос с ошибкой.

## Завершение запросов ввода/вывода

Когда драйвер завершает запрос ввода/вывода, он предоставляет статус завершения и количество байтов, переданных в запросе. Драйверы KMDF также могут указывать форсаж приоритета<sup>1</sup> для потока, издавшего запрос.

В табл. 8.16 приведен список методов завершения запросов ввода/вывода, которые могут вызывать драйверы WDF. Все методы UMDF принадлежат интерфейсу *IWDFIoRequest*.

---

<sup>1</sup> Совокупность системных параметров, позволяющая повысить уровень приоритета процесса на том или ином этапе. — Пер.

**Таблица 8.16.** Методы для завершения запросов ввода/вывода

Действие	Метод UMDF	Метод KMDF
Завершает запрос ввода/вывода, возвращая статус и форсаж приоритета (priority boost) по умолчанию	Complete	WdfRequestComplete
Завершает запрос ввода/вывода, возвращая статус, количество переданных байтов и форсаж приоритета по умолчанию	CompleteWithInformation	WdfRequestCompleteWithInformation
Завершает запрос ввода/вывода, возвращая статус, количество переданных байтов и форсаж приоритета	Никакой	WdfRequestCompleteWithPriorityBoost

Для завершения запроса ввода/вывода драйвер вызывает один из методов завершения ввода/вывода, приведенных в табл. 8.16. В запросах на создание, очистку и закрытие данные обычно не передаются, поэтому для их завершения драйвер может применить простой метод `Complete`. Но запросы на чтение, запись и большинство запросов IOCTL требуют, чтобы драйвер передавал данные. Поэтому для завершения любого запроса на передачу данных, драйвер должен использовать один из остальных методов, чтобы он мог известить приложение о количестве байтов, переданных в результате исполнения запроса.

Драйверы KMDF могут поставлять значение форсажа приоритета, завершая запрос методом `WdfRequestCompleteWithPriorityBoost`. Форсаж приоритета представляет собой системную константу, которая повышает приоритет времени исполнения потока, ожидающего завершения запроса ввода/вывода или получившего результаты завершенного запроса. Константы форсажа приоритета определены в файле `Wdm.h`. Если драйвер не указывает значение форсажа приоритета, то инфраструктура применяет значение по умолчанию, которое основывается на типе устройства, указанного драйвером при создании объекта устройства.

Список значений по умолчанию см. в разделе **Completing I/O Requests** (Завершение запросов ввода/вывода) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80614>.

В ответ на любой из методов завершения запроса, WDF завершает пакет IRP, на котором базируется запрос, после чего удаляет объект запроса WDF и все дочерние объекты. Если для объекта запроса WDF драйвер реализует функцию обратного вызова для очистки, WDF активирует обратный вызов этой функции перед тем, как завершить пакет IRP, с тем чтобы сам пакет IRP был действительным при исполнении функции обратного вызова. Так как пакет IRP остается действительным, то драйвер имеет доступ к его параметрам и буферам памяти.

После возвращения управления методом завершения ввода/вывода объект запроса ввода/вывода и его ресурсы освобождаются. После этого драйвер не должен предпринимать попыток обращения к объекту или какому-либо из его ресурсов, например, параметрам или буферам, которые были переданы в запросе.

Если запрос был отправлен драйверу очередью с последовательной диспетчеризацией, в результате вызова драйвером метода для завершения запроса инфраструктура может доставить ему следующий запрос в очереди. В случае очереди с параллельной диспетчеризацией инфраструктура может доставить драйверу следующий пакет в любое время. Если при вы-

зове метода завершения ввода/вывода драйвер удерживает какие-либо блокировки, необходимо убедиться в том, что методы обратного вызова по событию для очереди не блокируют те же самые объекты, т. к. это может вызвать взаимоблокировку. Но на практике это трудно выполнить, поэтому лучше всего не вызывать никаких методов завершения ввода/вывода, если удерживаются какие-либо блокировки.

## Отмененные и приостановленные запросы

По своей природе ввод/вывод Windows является асинхронным и реентерабельным. Система может потребовать от драйвера прекратить обработку запроса в любое время по множеству причин. Наиболее распространеными из этих причин являются следующие:

- ◆ поток или процесс, издавший запрос, отменяет его или прекращает свое исполнение;
- ◆ происходит системное событие Plug and Play или событие энергопотребления, например, переход в состояние гибернации;
- ◆ происходит удаление устройства или устройство уже было удалено.

Действия, предпринимаемые драйвером для прекращения обработки запроса ввода/вывода, зависят от причины приостановки или отмены выполнения запроса. Обычно драйвер может либо отменить запрос, либо завершить его с ошибкой. В некоторых ситуациях система может потребовать, чтобы драйвер приостановил (т. е. временно остановил) обработку запроса. Система позже извещает драйвер, когда возобновить выполнение.

Для удобства пользователей, драйверы должны отменять или приостанавливать с помощью функций обратных вызовов запросы ввода/вывода, которые занимают слишком долго в исполнении или которые могут не завершиться в установленный период времени, например, запросы для асинхронного ввода.

## Отмена запросов ввода/вывода

Действия, которые WDF предпринимает для отмены запроса ввода/вывода, зависят от того, был ли запрос уже доставлен целевому драйверу.

- ◆ Если запрос никогда не был доставлен драйверу — потому что инфраструктура еще не поставила его в очередь или потому, что он все еще находится в очереди, — инфраструктура отменяет или приостанавливает такой запрос автоматически, не извещая об этом драйвер.

Если первоначальный запрос ввода/вывода был отменен, инфраструктура завершает запрос, возвращая статус отмены.

- ◆ Если запрос был доставлен драйверу, но драйвер переслал его в другую очередь, инфраструктура отменяет запрос автоматически, не извещая драйвер.

Драйверы KMDF могут получить извещение об отмене, зарегистрировав обратный вызов функции `EvtIoCanceledOnQueue` для очереди.

Драйверы UMDF не могут получать такого рода извещения.

- ◆ Если запрос был доставлен драйверу и драйвер является его владельцем, инфраструктура не отменяет его.

Но если драйвер явно помечает запрос как отменяемый и регистрирует обратный вызов для отмены, инфраструктура извещает драйвер об отмене запроса. В своей функции об-

ратного вызова для отмены запроса драйвер должен завершить запрос или же предпринять необходимые меры для его быстрейшего завершения.

Чтобы пометить запрос как отменяемый или неотменяемый:

- ◆ драйверы UMDF вызывают метод `IWDFIoRequest::MarkCancelable` или `IWDFIoRequest::UnmarkCancelable` объекта запроса;
- ◆ драйверы KMDF вызывают метод `WdfRequestMarkCancelable` или `WdfRequestUnmarkCancelable`.

Когда драйвер помечает запрос как отменяемый, он передает функцию обратного вызова для отмены ввода/вывода, которую инфраструктура вызывает при отмене запроса.

Драйверы не должны оставлять запросы в неотменяемом состоянии на длительное время. Драйвер должен пометить запрос как отменяемый и зарегистрировать функцию обратного вызова для отмены ввода/вывода, если выполняется одно из следующих условий:

- ◆ запрос содержит длительную операцию;
- ◆ существует возможность, что запрос может быть никогда не исполненным, например, если запрос ожидает синхронный ввод.

Функция обратного вызова для отмены ввода/вывода должна выполнить все необходимые для отмены запроса действия, такие как, например, остановка любых выполняющихся операций ввода/вывода и отмена всех связанных запросов, которые уже были пересланы получателю ввода/вывода. В конечном счете, драйвер должен завершить запрос, возвращая одно из следующих значений статуса.

- ◆ Драйверы UMDF завершают запрос, возвращая статус `ERROR_OPERATION_ABORTED`, если запрос был отменен, прежде чем он был успешно завершен, или статус `S_OK`, если, несмотря на отмену, запрос был успешно завершен.
- ◆ Драйверы KMDF завершают запрос, возвращая статус `STATUS_CANCELLED`, если запрос был отменен, прежде чем он был успешно завершен, или статус `STATUS_SUCCESS`, если, несмотря на отмену, запрос был успешно завершен.

Запросы, отмеченные драйвером как отменяемые, нельзя пересылать в другую очередь. Прежде чем переслать запрос в другую очередь, драйвер должен пометить его как неотменяемый. После того, как запрос был поставлен в новую очередь, инфраструктура опять считает его отменяемым до тех пор, пока очередь не отправит его драйверу.

Чтобы правильно реализовать отмену запроса, необходимо уделять особое внимание возможности возникновения состояния гонок между веткой кода для нормального завершения запроса и веткой для его отмены. Помощь в выполнении отмены запросов может оказать применение предоставляемых инфраструктурой методов синхронизации.

Дополнительную информацию о применении методов синхронизации для отмены запросов см. в главе 10.

### **Альтернативный взгляд на отменяемость и неотменяемость**

Когда драйвер помечает запрос как отменяемый, он отдает право собственности над запросом процедуре отмены, которая может исполниться в любое время и завершить запрос. Процедура драйвера, завершающая запрос, должна получить обратно право собственности на запрос, прежде чем она может завершить запрос или переслать его в другую очередь. Драйвер возвращает себе право собственности над запросом от процедуры отмены с помощью методов, снимающих пометку отменяемости с запроса. Прежде чем драйвер может выполнять какие-либо операции с запросом, например, завершить его или переслать в другую очередь, он должен вызвать метод для снятия метки отменяемости с запроса и удостовериться в успешности его выполнения.

KMDF предоставляет драйверу способ для определения состояния отменяемости запроса, даже если драйвер не пометил его, как отменяемый. Если драйвер не пометил запрос как отменяемый, он может вызвать метод `WdfRequestIsCanceled`, чтобы проверить, не пытается ли менеджер ввода/вывода или первоначальный запрашивающий клиент отменить запрос. Такую проверку с помощью этого метода может выполнять драйвер, обрабатывающий данные на периодической основе. Например, драйвер, занимающийся обработкой изображений, может обрабатывать запрос на передачу данных небольшими фрагментами, проверяя на попытку отмены после обработки каждого фрагмента. В этом случае драйвер поддерживает отмену запроса ввода/вывода, но только после завершения обработки каждого отдельного фрагмента. Если драйвер определяет, что запрос был отменен, он исполняет все необходимые операции очистки и завершает запрос, возвращая статус `STATUS_CANCELLED`.

Глава 9 содержит информацию об отмене запросов ввода/вывода, которые драйвер отоспал получателю ввода/вывода.

Действия инфраструктуры при отмене запроса приводятся в табл. 8.17.

**Таблица 8.17. Действия инфраструктуры при отмене запросов ввода/вывода**

Обстоятельства отмены	Действия инфраструктуры
До того, как запрос был когда-либо доставлен драйверу	Отменяет запрос. Никаких действий со стороны драйвера не требуется, и драйвер не извещается об отмене
Когда запрос находится в очереди, но в какое-то время был доставлен драйверу. Это может быть ситуация, когда запрос ввода/вывода доставляется драйверу, который снова ставит его в очередь	Отменяет запрос. Никаких действий со стороны драйвера не требуется, и драйвер не извещается об отмене. Если драйвер KMDF зарегистрировал для очереди обратный вызов функции <code>EvtIoCanceledOnQueue</code> , то активирует этот вызов, после чего отменяет запрос. В противном случае отменяет запрос, как описано выше (Только в KMDF.)
Драйвер владеет запросом	Если драйвер пометил запрос как отменяемый, вызывает функцию обратного вызова для отмены, которую драйвер зарегистрировал для запроса. Если драйвер не пометил запрос как отменяемый, не предпринимает никаких действий. Драйвер может узнать, был ли запрос отменен, вызвав метод <code>WdfRequestIsCanceled</code> (Только в KMDF.)

### Совет

В главе 24 описывается, каким образом вставить комментарии в функцию обратного вызова драйвера, чтобы инструмент SDV мог проанализировать ее на соответствие требованиям правил KMDF для завершения и отмены запросов.

## Приостановка запроса ввода/вывода

Когда устройство переходит в состояние пониженного энергопотребления (часто потому что пользователь указал режим гибернации или закрыл крышку ноутбука), драйвер может завершить, поставить в другую очередь или продолжать удерживать запросы, которые он в настоящее время обрабатывает.

Драйвер может реализовать обратный вызов функции остановки ввода/вывода для извещения о таких изменениях в состоянии энергопотребления. А именно:

- ◆ UMDF извещает драйвер о предстоящем изменении в энергопотреблении, вызывая метод `IQueueCallbackIoStop::OnIoStop` объекта обратного вызова очереди для каждого такого запроса;
- ◆ KMDF извещает драйвер о предстоящем изменении энергопотребления, вызывая функцию обратного вызова `EvtIoStop` для каждого такого запроса.

Каждый вызов содержит флаги, которые указывают причину остановки очереди и текущее состояние отменяемости запроса ввода/вывода. В зависимости от значений этих флагов драйвер может предпринять следующее:

- ◆ завершить запрос;
- ◆ поставить запрос в другую очередь;
- ◆ игнорировать извещение, если текущий запрос будет выполнен вовремя;
- ◆ подтвердить получение извещения, но продолжать удерживать запрос (только в KMDF).

Если очередь останавливается по причине удаления устройства, драйвер должен завершить запрос как можно быстрее. После того как инфраструктура вызвала функции обратного вызова для остановки ввода/вывода, процесс удаления устройства блокируется до тех пор, пока не завершены все запросы, принадлежащие драйверу.

Драйверы должны реализовывать обратные вызовы функций остановки ввода/вывода для любого запроса, завершение которого может занять длительное время или который может не завершиться совсем. Например, если драйвер пересыпает запросы получателю ввода/вывода, то эти запросы могут еще некоторое время ожидать обработки на получателе. Поэтому необходимо реализовать обратный вызов функции остановки ввода/вывода для очереди, которая отправила запрос драйверу. Управление запросами ввода/вывода при подготовке к переходу устройства в режим пониженного энергопотребления способствует удобству пользователя при работе с ноутбуками и другими системами с управляемым энергопотреблением.

## Адаптивные тайм-ауты в UMDF

Нормативы Windows для завершения и отмены ввода/вывода требуют, чтобы драйверы:

- ◆ поддерживали отмену для запросов ввода/вывода, завершение которых может занять неопределенное время;
- ◆ завершали запросы ввода/вывода в течение разумного периода времени (обычно 10 секунд или меньше) после отмены;
- ◆ не блокировали потоки ввода/вывода на необоснованно длительное время при исполнении ввода/вывода. Потоки ввода/вывода UMDF являются ограниченным ресурсом, поэтому блокирование потока на протяжении длительного времени может понизить производительность драйвера.

С целью способствования драйверам пользовательского режима соблюдать эти нормативы, UMDF поддерживает адаптивные тайм-ауты. UMDF отслеживает ход критических операций ввода/вывода, задержка исполнения которых может замедлить систему. К критическим операциям относятся запросы на очистку, закрытие, отмену, Plug and Play и управления энергопотреблением.

Когда отражатель передает критический запрос хост-процессу драйвера, он наблюдает за ходом операций ввода/вывода. Пока драйвер пользовательского режима не завершит критический запрос, он должен выполнять по одной операции через равные промежутки времени. Например, если устройство переходит в режим пониженного энергопотребления, функции обратного вызова драйвера должны или возвратить управление до окончания периода тайм-аута или же завершить через равные промежутки времени все ожидающие выполнения запросы ввода/вывода, которые нельзя оставить незаконченными, чтобы отражатель продолжил период тайм-аута.

Если период тайм-аута истекает, отражатель завершает хост-процесс и извещает о проблеме через систему WER (Windows Error Reporting, докладывание об ошибках Windows). В настоящее время период тайм-аута по умолчанию равняется одной минуте. Если драйверу необходимо выполнять операции, для завершения которых требуется длительное время, он должен обрабатывать их асинхронно, в отдельном потоке или в пользовательском рабочем элементе.

## **Самоуправляемый ввод/вывод**

Хотя для большинства драйверов рекомендуется применять поддержку ввода/вывода и Plug and Play,строенную в WDF, некоторые драйверы выполняют операции ввода/вывода, не связанные с организованными в очередь запросами ввода/вывода, не подлежащие управлению энергопотреблением, или которые необходимо синхронизировать с действиями драйверов WDM в том же самом стеке устройств. Например, образец драйвера Pcidrv использует обратные вызовы самоуправляемого ввода/вывода для запуска и остановки таймера WDT. Подобным образом драйверу может быть необходимо взаимодействовать со своим устройством или другим драйвером в определенной точке последовательности операций запуска или остановки устройства. Для выполнения таких требований WDF предоставляет возможность самоуправляемого ввода/вывода.

Обратные вызовы самоуправляемого ввода/вывода соответствуют более близко находящимся в стеке пакетам IRP WDM для Plug and Play и управления энергопотребления, чем другие обратные вызовы WDF для Plug and Play и управления энергопотреблением.

### **Самоуправляемый ввод/вывод и машина состояний энергопотребления**

Самоуправляемый ввод/вывод представляет собой универсальное средство в машине состояний энергопотребления. Все другие обратные вызовы имеют четкие контракты и указывают, какие действия должны в них выполняться. А самоуправляемый ввод/вывод покрывает все прочее, что не попадает под условия однозначных контрактов. (Даун Холэн (*Down Holan*), команда разработчиков Windows Driver Foundation, Microsoft.)

Для использования самоуправляемого ввода/вывода драйвер реализует функции обратного вызова по событию самоуправляемого ввода/вывода. Инфраструктура вызывает эти функции обратного вызова при изменении состояний Plug and Play и энергопотребления, когда устройство добавляется или удаляется из системы, когда устройство останавливается для перераспределения ресурсов, когда простоявшее устройство переходит в режим пониженного энергопотребления и когда устройство возвращается в рабочее состояние из состояния пониженного энергопотребления.

Функции обратного вызова самоуправляемого ввода/вывода прямым образом соответствуют изменениям состояний Plug and Play и энергопотребления. При вызове этим функциям передается только дескриптор объекта устройства и никаких других параметров. Если драйвер

регистрирует такие функции обратного вызова, то инфраструктура вызывает их в назначенное время, чтобы драйвер мог выполнить все необходимые действия.

Подробная информация о порядке вызова инфраструктурой этих функций изложена в главе 7.

- ◆ Для использования самоуправляемого ввода/вывода драйверы UMDF реализуют интерфейс `IPnpCallbackSelfManagedIo` на объекте обратного вызова устройства.
- ◆ Драйверы KMDF реализуют функции семейства `EvtDeviceSelfManagedIoXxx` и регистрируют эти функции, вызывая метод `WdfDeviceInitSetPnpPowerEventCallbacks` перед созданием объекта устройства.

В табл. 8.18 перечислены методы самоуправляемого ввода/вывода для UMDF и KMDF, с указанием обстоятельств, при которых они вызываются.

**Таблица 8.18. Методы самоуправляемого ввода/вывода**

Метод UMDF в интерфейсе <code>IPnpCallbackSelfManagedIo</code>	Метод KMDF	Когда вызывается
<code>OnSelfManagedIoCleanup</code>	<code>EvtDeviceSelfManagedIoCleanup</code>	Непосредственно перед удалением объекта устройства
<code>OnSelfManagedIoFlush</code>	<code>EvtDeviceSelfManagedIoFlush</code>	При обработке PnP-запроса <code>IRP_MN_REMOVE_DEVICE</code> или <code>IRP_MN_SURPRISE_REMOVAL</code> , после удаления управляемых энергопотреблением очередей. Эта функция должна завершать неудачей исполнение любых запросов ввода/вывода, которые драйвер не завершил до удаления устройства
<code>OnSelfManagedIoInit</code>	<code>EvtDeviceSelfManagedIoInit</code>	При запуске устройства, после возвращения управления функцией обратного вызова драйвера для входа в состояние D0, но перед тем, как WDF завершит пакет IRP. Вызывается только при первоначальном запуске, но не при возвращении устройства в рабочее состояние из состояния пониженного энергопотребления
<code>OnSelfManagedIoRestart</code>	<code>EvtDeviceSelfManagedIoRestart</code>	Когда устройство возвращается в рабочее состояние из состояния пониженного энергопотребления. Вызывается, только если WDF раньше вызвала метод драйвера для пристановки самоуправляемого ввода/вывода
<code>OnSelfManagedIoStop</code>	Никакой	В настоящее время не вызывается
<code>OnSelfManagedIoSuspend</code>	<code>EvtDeviceSelfManagedIoSuspend</code>	Перед вызовом инфраструктурой WDF любой из функций обратного вызова для событий Plug and Play или энергопотребления в последовательности операций выключения устройства, при наличии любого из следующих обстоятельств: <ul style="list-style-type: none"> <li>• устройство намеревается перейти в состояние пониженного энергопотребления;</li> <li>• выполняется удаление устройства или устройство было неожиданно удалено;</li> <li>• менеджер PnP подготавливается к перераспределению системных аппаратных ресурсов среди устройств системы</li> </ul>

## **Самоуправляемый ввод/вывод при запуске и перезапуске устройства**

При первоначальной загрузке системы или когда пользователь вставляет устройство в разъем, WDF вызывает функцию обратного вызова драйвера для инициализации самоуправляемого ввода/вывода (т. е. метод `IPnpCallbackSelfmanagedIo::OnSelfManagedIoInit` или функция `EvtDeviceSelfManagedIoInit`) после возврата управления функцией обратного вызова драйвера для входа в состояние D0, но перед тем, как WDF завершит нижележащий в стеке пакет IRP. Функции инициализации самоуправляемого ввода/вывода вызываются только при первоначальном запуске, но не при возвращении устройства в рабочее состояние из состояния пониженного энергопотребления.

Функции обратного вызова для инициализации самоуправляемого ввода/вывода должны выполнять все необходимые действия для инициализации ввода/вывода, которые не выполняются инфраструктурой. Например, драйвер, которому требуется отслеживать состояние своего устройства, может инициализировать и запустить таймер. Драйвер также может выполнить одноразовую инициализацию, для которой требуется электропитание, например перечисление статических дочерних устройств на основе версии аппаратного обеспечения.

Когда устройство возвращается в рабочее состояние из состояния пониженного энергопотребления (в которое оно было переведено, например, по причине простоя или для перераспределения ресурсов), WDF вызывает функцию обратного вызова для перезапуска самоуправляемого ввода/вывода.

Подобно функции обратного вызова для инициализации самоуправляемого ввода/вывода, функция обратного вызова для перезапуска является последней функцией, вызываемой после возвращения устройства в рабочее состояние, но перед тем, как WDF завершит пакет IRP, активировавший переход в рабочее состояние. Функция обратного вызова для перезапуска должна возобновить все операции ввода/вывода, начатые функцией обратного вызова для инициализации самоуправляемого ввода/вывода и позже приостановленные при выходе устройства из рабочего состояния. Обычно это означает, что она отменяет действия, выполненные функцией обратного вызова для приостановки самоуправляемого ввода/вывода.

При возобновлении работы устройства функция обратного вызова для перезапуска вызывается, только если работа устройства была приостановлена функцией обратного вызова для приостановки работы. То есть, функция обратного вызова для перезапуска вызывается только при возвращении устройства в рабочее состояние из состояния пониженного энергопотребления, в которое оно было переведено по причине простоя или для перераспределения ресурсов. При первоначальном физическом подключении устройства пользователем она не вызывается.

## **Самоуправляемый ввод/вывод при переходе устройства в состояние пониженного энергопотребления или при его удалении**

При переходе устройства в состояние пониженного энергопотребления или при его удалении WDF вызывает одну или несколько функций обратного вызова для самоуправляемого ввода/вывода, чтобы драйвер мог остановить операции самоуправляемого ввода/вывода и выполнить необходимую очистку.

Всякий раз, когда устройство выполняет последовательность перехода в состояние пониженного энергопотребления — будь то по причине его простоя, удаления или для перерас-

пределения системных ресурсов — WDF вызывает функцию обратного вызова для приостановки самоуправляемого ввода/вывода. Эта функция должна остановить все выполняющиеся операции самоуправляемого ввода/вывода, для выполнения которых требуется присутствие устройства. При перераспределении ресурсов, переходе в режим пониженного энергопотребления и запланированного удаления эта функция вызывается, пока устройство все еще находится в рабочем состоянии энергопотребления (D0). При неожиданном удалении устройства эта функция вызывается перед функцией обратного вызова для извещения о неожиданном удалении, если устройство находилось в состоянии пониженного энергопотребления, и после нее, если устройство находилось в рабочем состоянии.

При удалении устройства WDF вызывает функцию обратного вызова для очистки самоуправляемого ввода/вывода после остановки устройства, когда оно больше не находится в состоянии D0. Эта функция должна завершать неудачей любые запросы ввода/вывода, которые драйвер не завершил до удаления устройства. Она вызывается после возвращения управления функцией обратного вызова для приостановки самоуправляемого ввода/вывода и функциями для выхода из состояния D0.

Наконец, после того, как устройство было удалено и при удалении объекта устройства, WDF вызывает функцию обратного вызова для очистки самоуправляемого ввода/вывода. Эта функция должна обеспечить, что вся деятельность самоуправляемого ввода/вывода полностью остановлена, и должна освободить любые ресурсы, выделенные самоуправляемому вводу/выводу функцией обратного вызова для инициализации самоуправляемого ввода/вывода. Функция очистки вызывается только один раз.

Для объекта PDO инфраструктура не вызывает функцию обратного вызова *EvtDeviceSelfManagedIoCleanup* до тех пор, пока сам объект PDO не будет удален, когда устройство или физически отключено или когда удален его родитель. Если инфраструктура не удаляет объект PDO и устройство позже перезапускается, инфраструктура вызывает функцию *EvtDeviceSelfManagedIoRestart*, не вызывая предварительно функцию *EvtDeviceSelfManagedIoCleanup*.

Подробное объяснение последовательности обратных вызовов функций, принимающих участие в переводе устройства в состояние пониженного энергопотребления и его удалении, изложено в главе 7.

## Пример KMDF: реализация таймера WDT

В образце драйвера Pcidrv самоуправляемый ввод/вывод применяется для реализации таймера WDT, который используется при обнаружении аппаратного соединения и для проверок на наличие зависаний. Для реализации таймера WDT-драйверу требуется выполнить следующие задачи:

- ◆ организовать обратные вызовы функций для событий самоуправляемого ввода/вывода;
- ◆ инициализировать таймер в функции *EvtDeviceSelfManagedIoInit*;
- ◆ остановить таймер в функции *EvtDeviceSelfManagedIoSuspend*;
- ◆ перезапустить таймер в функции *EvtDeviceSelfManagedIoRestart*;
- ◆ удалить таймер и ресурсы в функции *EvtDeviceSelfManagedIoCleanup*.

Образец драйвера Pcidrv не реализует функцию обратного вызова *EvtDeviceSelfManagedIoFlush*, т. к. его самоуправляемый ввод/вывод не содержит запросов ввода/вывода. Достаточно лишь функций обратного вызова для приостановки устройства и очистки ресурсов.

Драйвер регистрирует свои функции обратного вызова для самоуправляемого ввода/вывода, устанавливая их точки входа в структуре `WDF_PNP_POWER_CALLBACKS`, вместе с другими функциями обратного вызова для событий Plug and Play и энергопотребления, включая такие функции, как, например, `EvtDeviceD0Entry` и `EvtDeviceD0Exit`. Регистрация выполняется драйвером в функции обратного вызова `EvtDriverDeviceAdd`, перед созданием им объекта `WDFDEVICE`.

### Пример: регистрация функций обратного вызова для самоуправляемого ввода/вывода

Драйвер `Pcidrv` регистрирует эти функции обратного вызова в функции `PciDrvEvtDeviceAdd` из файла `Pcidrv.c`, как показано в листинге 8.18.

#### Листинг 8.18. Регистрация функций обратного вызова для самоуправляемого ввода/вывода в драйвере KMDF

```
WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;
// Инициализация структуры pnpPowerCallbacks.
WDF_PNPPOWER_EVENT_CALLBACKS_IN rrr(&pnpPowerCallbacks);
// Установка точек входа для функций обратного вызова
// для самоуправляемого ввода/вывода.
pnpPowerCallbacks.EvtDeviceSelfManagedIoInit =
    Pci_DrvEvtDeviceSelfManagedIoInit;
pnpPowerCallbacks.EvtDeviceSelfManagedIoCleanup =
    PciDrvEvtDeviceSelfManagedIoCleanup;
pnpPowerCallbacks.EvtDeviceSelfManagedIoSuspend =
    PciDrvEvtDeviceSelfManagedIoSuspend;
pnpPowerCallbacks.EvtDeviceSelfManagedIoRestart =
    PciDrvEvtDeviceSelfManagedIoRestart;
// Регистрация функций обратного вызова для событий
// PnP и энергопотребления.
WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);
```

Как можно видеть в предыдущем примере, драйвер `Pcidrv` регистрирует обратные вызовы функций `EvtDeviceSelfManagedIoInit`, `EvtDeviceSelfManagedIoCleanup`, `EvtDeviceSelfManagedIoSuspend` и `EvtDeviceSelfManagedIoRestart`.

### Пример: создание и инициализация таймера

Драйвер `Pcidrv` создает, инициализирует и запускает таймер WDT в своей функции обратного вызова `EvtDeviceSelfManagedIoInit`. Таймер WDT является объектом таймера WDF. Когда таймер завершает свою работу, KMDF ставит в очередь процедуру отложенного вызова, которая вызывает функцию драйвера `EvtTimerFunc`.

В листинге 8.19 показан исходный код функции обратного вызова `EvtDeviceSelfManagedIoInit` образца драйвера `Pcidrv`, которая находится в файле `Pcidrv.c`.

#### Листинг 8.19. Инициализация самоуправляемого ввода/вывода в драйвере KMDF

```
NTSTATUS PciDrvEvtDeviceSelfManagedIoInit(IN WDFDEVICE Device)
{
    PFDO_DATA fdoData = NULL;
```

```

WDF_TIMER_CONFIG          wdfTimerConfig;
NTSTATUS                  status;
WDF_OBJECT_ATTRIBUTES     timerAttributes;
PACED_CODE();

fdoData = FdoCetData(Device);
// Создаем процедуру отложенного вызова для запуска по таймеру
// для обнаружения соединения и для проверки на наличие
// зависшего оборудования.
WDF_TIMER_CONFIG_INIT(&wdfTimerConfig, NICWatchDogEvtTimerFunc);
WDF_OBJECT_ATTRIBUTES_INIT(&timerAttributes);
timerAttributes.ParentObject = fdoData->WdfDevice;
status = WdfTimerCreate(&wdfTimerConfig,
                       &timerAttributes,
                       &fdoData->WatchDogTimer);
if (!NT_SUCCESS(status)) {
    return status;
}
NICStartWatchDogTimer(fdoData);
return status;
}

```

Драйвер объявляет две структуры для создания таймера: структуру типа `WDF_TIMER_CONFIG` с именем `wdfTimerConfig` и структуру типа `WDF_OBJECT_ATTRIBUTES` с именем `timerAttributes`. Драйвер инициализирует структуру `wdfTimerConfig` с помощью функции `WDF_TIMER_CONFIG_INIT`, передавая ей в качестве параметров саму структуру `wdfTimerConfig` и указатель на функцию `NICWatchdogEvtTimerFunc`, которая является обратным вызовом `EvtTimerFunc` драйвера.

Потом драйвер инициализирует структуру атрибутов с помощью функции `WDF_OBJECT_ATTRIBUTES_INIT`. Значение поля `ParentObject` устанавливается в объект устройства, чтобы KMDF удалила таймер при удалении его объекта устройства.

Наконец, драйвер создает таймер, вызывая метод `WdfTimerCreate` и передавая ему в качестве параметров структуру конфигурации, структуру атрибутов и адрес для получения дескриптора таймера. При успешном создании таймера инфраструктурой KMDF драйвер `Pcidrv` запускает таймер, вызывая внутреннюю функцию `NICStartWatchDogTimer`.

### Пример: запуск таймера

В листинге 8.20 приводится функция `NICStartWatchDogTimer` драйвера для запуска таймера. Сама функция определена в исходном файле `Sys\Hw\Isrdpc.c`.

#### Листинг 8.20. Запуск таймера WDT в драйвере KMDF

```

VOID NICStartWatchDogTimer(IN PFDO_DATA  FdoData)
{
    LARCE_INTECER dueTime;
    if (!FdoData->CheckForHang) {
        // Устанавливается флаг обнаружения соединения.
        MP_SET_FLAC(FdoData, fMP_ADAPTER_LINK_DETECTION);
        FdoData->CheckForHang = FALSE;
        FdoData->bLinkDetectionWait = FALSE;
    }
}

```

```

FdoData->bLookForLink = FALSE;
dueTime.QuadPart = NIC_LINK_DETECTION_DELAY;
}
else {
    dueTime.QuadPart = NIC_CHECK_FOR_HANG_DELAY;
}
WdfTimerStart(FdoData->WatchDogTimer, dueTime.QuadPart);
return;
}

```

Функция присваивает таймеру аппаратно-зависимое значение, зависящее от того, пытается ли драйвер обнаружить соединение или проверяет, не зависло ли устройство. После этого функция запускает таймер, вызывая метод `WdfTimerStart`. Когда время таймера истекает, KMDF ставит в очередь процедуру отложенного вызова (DPC), которая активирует функцию таймера `NICWatchDogEvtTimerFunc` драйвера. Функция таймера выполняет требуемую задачу — обнаружение соединения или проверку на зависание — после чего перезапускает таймер, вызывая метод `WdfTimerStart` точно таким же образом, как показано в листинге 8.19.

### Пример: остановка таймера

Когда устройство покидает рабочее состояние или удаляется из системы, KMDF вызывает функцию обратного вызова `EvtDeviceSelfManagedIoSuspend`. В образце драйвера `Pcidrv` эта функция останавливает таймер, как показано в листинге 8.21.

#### Листинг 8.21. Приостановка самоуправляемого ввода/вывода в драйвере KMDF

```

NTSTATUS PciDrvEvtDeviceSelfManagedIoSuspend(IN WDFDEVICE Device)
{
    PFDO_DATA fdoData = NULL;
    PACED_CODE();
    fdoData = FdoCetData(Device);
    // Останавливаем таймер и ожидаем завершения исполнения
    // процедуры отложенного вызова.
    WdfTimerStop(fdoData->WatchDogTimer, TRUE);
    return STATUS_SUCCESS;
}

```

Для остановки таймера драйвер просто вызывает метод `WdfTimerStop`, передавая ему в качестве параметров дескриптор таймера и булево значение. Драйвер `Pcidrv` передает значение `TRUE`, таким образом указывая, что если очередь процедур DPC содержит процедуру отложенного вызова таймера `NICWatchDogEvtTimerFunc` или любые другие процедуры отложенного вызова, созданные драйвером, инфраструктура должна дождаться возвращения управления всеми этими процедурами, прежде чем останавливать таймер. Значение `FALSE` в этом параметре означает, что KMDF должна остановить таймер немедленно, не ожидая завершения никаких процедур отложенного вызова.

Метод `WdfTimerStop` возвращает `TRUE`, если объект таймера находился в очереди таймера системы. Но драйвер `Pcidrv` не проверяет возвращаемое значение, т. к. он ожидает завершения исполнения всех процедур DPC драйвера, так что обстоятельство, был ли таймер уже остановлен, не является важным.

## Пример: перезапуск таймера

Когда устройство возвращается в рабочее состояние из состояния пониженного энергопотребления, KMDF активирует обратный вызов функции *EvtDeviceSelfManagedIoRestart*. В драйвере Pcidrv этот обратный вызов перезапускает таймер, как показано в листинге 8.22. Этот пример функции находится в исходном файле Pcidrv.c.

### Листинг 8.22. Перезапуск самоуправляемого ввода/вывода в драйвере KMDF

```
NTSTATUS PciDrvEvtDeviceSelfManagedIoRestart(IN WDFDEVICE Device)
{
    PFDO_DATA fdoData;
    PACED_CODE();
    fdoData = FdoCetData(Device);
    // Перезапускаем таймер WDT.
    NICStartWatchDogTimer(fdoData);
    return STATUS_SUCCESS;
}
```

Для перезапуска таймера необходимо всего лишь вызвать внутреннюю функцию *NICStartWatchDogTimer*, как было описано ранее. Так как объект устройства и таймер, который является дочерним объектом объекта устройства, не были удалены, когда устройство покинуло рабочее состояние, то драйверу не требуется снова выполнять инициализацию или создавать объект таймера.

## Пример: удаление таймера

Когда устройство удаляется, драйвер удаляет таймер с помощью функции *EvtDeviceSelfManagedIoCleanup*, как показано в листинге 8.23.

### Листинг 8.23. Удаление самоуправляемого ввода/вывода в драйвере KMDF

```
VOID PciDrvEvtDeviceSelfManagedIoCleanup(IN WDFDEVICE Device)
{
    PFDO_DATA fdoData = NULL;
    PACED_CODE();
    fdoData = FdoCetData(Device);
    if (fdoData->WatchDogTimer) {
        WdfObjectDelete(fdoData->WatchDogTimer);
    }
    return;
}
```

Для удаления таймера драйвер вызывает метод *WdfObjectDelete*, передавая ему в параметрах указатель на объект таймера. Если драйвер выделил таймеру какие-либо дополнительные ресурсы, они также освобождаются в этой функции. Вызов метода *WdfObjectDelete* не является обязательным, т. к. инфраструктура автоматически удаляет объект таймера при удалении его родительского объекта типа *WDFDEVICE*.

# ГЛАВА 9

## Получатели ввода/вывода

Большинство драйверов завершают только некоторые запросы ввода/вывода и пересыпают другие запросы вниз по стеку устройств. Драйверы также могут сами издавать запросы ввода/вывода. Независимо от того, создает ли драйвер сам новые запросы или просто передает вниз по стеку запросы, которые он получает от инфраструктуры, пунктом назначения этих запросов является *получатель (target) ввода/вывода*.

В начале этой главы приводится основная информация о получателях ввода/вывода. Далее следует описание, каким образом драйвер может создавать и форматировать запросы ввода/вывода и посыпать их получателю ввода/вывода.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Echo_driver	%wdk%\Src\Umdf\Usb\Echo_driver
Filter	%wdk%\Src\Umdf\Usb\Filter
Fx2_Driver	%wdk%\Src\Umdf\Usb\Fx2_driver
Kbfiltr	%wdk%\Src\Kmdf\Kbfiltr\Sys
Ndisedge	%wdk%\Src\Kmdf\Ndisedge\60
Osrusbf2	%wdk%\Src\Kmdf\Osrusbf2\Sys\Final
Toastmon	%wdk%\Src\Kmdf\Toaster\Toastmon
Usbsamp	%wdk%\Src\Kmdf\Usbsamp
<b>Документация WDK</b>	
ACCESS_MASK	<a href="http://go.microsoft.com/fwlink/?LinkId=80616">http://go.microsoft.com/fwlink/?LinkId=80616</a>
Handling I/O Requests in Framework-based Drivers <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80613">http://go.microsoft.com/fwlink/?LinkId=80613</a>
Object Names <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80615">http://go.microsoft.com/fwlink/?LinkId=80615</a>
Specifying WDF Directives <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82953">http://go.microsoft.com/fwlink/?LinkId=82953</a>
WINUSB_SETUP_PACKET	<a href="http://go.microsoft.com/fwlink/?LinkId=83355">http://go.microsoft.com/fwlink/?LinkId=83355</a>

<sup>1</sup> Обработка запросов ввода/вывода в драйверах на основе инфраструктуры (WDF). — *Пер.*

<sup>2</sup> Имена объектов. — *Пер.*

## О получателях ввода/вывода

Драйвер WDF использует получатель ввода/вывода для того, чтобы послать запрос ввода/вывода другому драйверу. Получатель ввода/вывода является собой объект WDF, который представляет объект устройства, которому направлен запрос ввода/вывода. Получателем ввода/вывода может быть драйвер UMDF, драйвер KMDF, драйвер WDM или любой другой драйвер режима ядра.

Но получатель ввода/вывода — это что-то больше, нежели простой указатель на объект устройства. Каждый получатель ввода/вывода поддерживает методы для исполнения следующих операций:

- ◆ форматирование запросов на чтение, запись и IOCTL, как этого требует получатель;
- ◆ определение состояния Plug and Play получателя, если получатель представляет объект устройства Plug and Play;
- ◆ опрашивание получателя и управления работой получателя и потоком запросов ввода/вывода к получателю.

По умолчанию WDF посыпает запросы ввода/вывода только получателям ввода/вывода, пребывающим в рабочем состоянии, с тем, чтобы драйвер не посыпал запросов ввода/вывода получателю, который был остановлен или удален. Драйвер может изменить эту стандартную установку для получателя.

Драйвер может посыпать запросы ввода/вывода получателю ввода/вывода для синхронного или асинхронного завершения, а также может предоставлять функции обратного вызова для завершения асинхронных запросов.

Драйвер KMDF также способен предоставлять функции обратного вызова, посредством которых драйвер может запросить извещение об изменениях в состоянии Plug and Play для планового или неожиданного удаления удаленного устройства получателя.

### Совет

Драйверы WDF применяют получатель ввода/вывода во всех ситуациях, в каких драйверы WDM используют процедуру `IoCallDriver`.

## Стандартные получатели ввода/вывода

Стандартным получателем ввода/вывода является следующий нижележащий драйвер в стеке устройств. Для передачи запроса вниз по стеку драйверы WDF используют стандартный получатель ввода/вывода. Когда драйвер вызывает инфраструктуру, чтобы создать объект устройства, инфраструктура создает и инициализирует стандартный получатель ввода/вывода для объекта устройства. Впоследствии, драйвер вызывает инфраструктурный метод для получения доступа к объекту получателя ввода/вывода. Объект получателя ввода/вывода является потомком объекта устройства. Инфраструктура создает стандартный получатель ввода/вывода только для функциональных драйверов и для драйверов фильтра.

## Удаленные получатели ввода/вывода в драйверах KMDF

Удаленный получатель ввода/вывода представляет любого получателя запроса ввода/вывода, иного, чем следующий нижележащий объект устройства. Драйверы KMDF ис-

<sup>1</sup> Задание директив WDF. — *Пер.*

пользуют удаленных получателей ввода/вывода для передачи запросов ввода/вывода другим стекам устройств или вверх по собственному стеку устройств. Важным различием между локальными и удаленными получателями ввода/вывода является то обстоятельство, что запросы, переданные удаленному получателю ввода/вывода, больше не продвигаются вниз по текущему стеку устройств.

Например, в некоторых ситуациях драйвер может нуждаться в информации от другого стека устройств, прежде чем он может завершить запрос ввода/вывода. В таком случае драйвер создает удаленного получателя ввода/вывода и запрос ввода/вывода, после чего отправляет запрос получателю. Для удаленного получателя ввода/вывода инфраструктура направляет запрос наверх стека устройств.

Драйвер может также использовать удаленный получатель ввода/вывода для того, чтобы отправить запрос IOCTL наверх своего собственного стека устройств, с тем, чтобы все драйверы в стеке могли иметь возможность обработать запрос. Если драйвер отправляет запрос стандартному получателю ввода/вывода, только драйверы, расположенные в стеке ниже его, получают этот запрос. Чтобы отправить такой запрос, драйвер получает указатель на `DEVICE_OBJECT` WDM наверху своего стека устройств, вызывая процедуру `IoGetAttachedDeviceReference` менеджера ввода/вывода, после чего создает удаленный получатель ввода/вывода, используя полученный указатель.

## Общие и специализированные получатели ввода/вывода

Кроме деления получателей ввода/вывода на стандартные и удаленные, WDF также категоризирует их как общие и специализированные. Общий получатель ввода/вывода может представлять объект устройства для любого типа устройства. Драйвер, который отправляет запрос ввода/вывода, является ответственным за форматирование запроса в таком виде, в каком получатель ожидает получить его. Поэтому драйвер должен заполнить все специфичные для типа устройства контрольные блоки и выполнить прочие необходимые подготовительные операции. Общие получатели ввода/вывода не поддерживают никаких особых, специфичных для устройства, форматов данных.

Специализированные получатели ввода/вывода предоставляют форматирование данных, специфичное определенному типу устройства, например специфичные для типа устройства блоки запроса. Если инфраструктура реализует специализированные получатели ввода/вывода, поддерживающие формат данных устройства, драйвер должен использовать их. Для некоторых типов устройств драйверу может требоваться использовать структуры WDM для того, чтобы должным образом форматировать запросы ввода/вывода для определенного устройства.

Специализированные получатели ввода/вывода дают возможность использовать расширения для WDF, поддерживающие новое аппаратное обеспечение и типы протоколов. В результате не требуется организовывать базовую поддержку для новых промышленных стандартов на уровне отдельных драйверов. Вместо этого, компания Microsoft может добавить такую поддержку в WDF, где она может быть протестирована и предоставлена всем драйверам.

WDF поддерживает специализированные получатели ввода/вывода для устройств USB как в KMDF, так и в UMDF. Для взаимодействия с USB-получателями ввода/вывода инфраструктура WDF использует блоки URB. Интерфейсы получателей ввода/вывода USB содержат методы для создания и отправки блоков URB, так что драйверу не требуется делать этого самому.

UMDF также поддерживает специализированные получатели ввода/вывода для дескрипторов файлов. Дополнительную информацию по этому вопросу см. в разд. "Получатели ввода/вывода *FileHandle* в драйверах UMDF" далее в этой главе.

## Реализация получателя ввода/вывода в UMDF

UMDF реализует получателей ввода/вывода посредством следующих двух специальных механизмов:

- ◆ диспетчеров ввода/вывода, которые отправляют запросы ввода/вывода из стека устройств UMDF соответствующей подсистеме;
- ◆ файлов получателей ввода/вывода, которые представляют сеансы ввода/вывода для получателей ввода/вывода.

С помощью диспетчера ввода/вывода UMDF может сопоставить общий запрос ввода/вывода на чтение, запись или IOCTL с вызовом специфичной функции для получателя ввода/вывода. Например, диспетчер USB получает общий запрос на чтение и, в свою очередь, вызывает специфичную функцию из WinUSB.dll. Драйверы UMDF не должны вызывать функции WinUSB прямым образом, т. к. такие запросы ввода/вывода немедленно покидают текущий стек устройств и поступают в другие подсистемы. Инфраструктура не может перехватить такой запрос, чтобы обеспечить нижележащим драйверам в стеке возможность обработать его.

Кроме этого, UMDF требует дескриптор файла (и, соответственно, сеанс) с каждым запросом. Файлы получателей ввода/вывода представляют сеансы для объектов получателей ввода/вывода.

Для драйверов KMDF не требуются ни планировщики ввода/вывода, ни файлы получателей ввода/вывода, т. к. драйверы режима ядра могут использовать пакеты IRP для взаимодействия непосредственно через менеджер ввода/вывода. Пакет IRP представляет собой единообразный интерфейс, используемый всеми стеками устройств режима ядра.

### Диспетчеры ввода/вывода UMDF

Диспетчеры ввода/вывода UMDF направляют запросы ввода/вывода получателям ввода/вывода, находящимся вне стека устройств пользовательского режима. Диспетчер ввода/вывода получает запросы ввода/вывода, которые дошли до самого низа стека устройств пользовательского режима, и решает, каким образом отправить эти запросы стеку устройств режима ядра. Например, диспетчер USB направляет запросы в библиотеку пользовательского режима WinUSB.dll, которая затем направляет их в стек устройств режима ядра для устройства USB.

UMDF не поддерживает удаленных получателей ввода/вывода, поэтому драйверы UMDF могут отправлять запросы ввода/вывода только следующему нижележащему драйверу. Каждый запрос ввода/вывода доходит до низа стека устройств режима ядра, если только драйвер UMDF не завершит его. Планировщик играет роль соединителя между низом стека устройств UMDF и стандартным получателем ввода/вывода, которым является объект устройства Down вверху стека устройств режима ядра.

Когда драйвер UMDF посылает запрос ввода/вывода получателю ввода/вывода, инфраструктура создает пакет запроса ввода/вывода и отправляет его стандартному получателю ввода/вывода. Когда пакет запроса доходит до низа стека устройств UMDF, диспетчер вво-

да/вывода принимает пакет и преобразовывает его в соответствующие функции API для подсистемы получателя.

UMDF реализует несколько диспетчеров ввода/вывода нескольких типов (рис. 9.1).

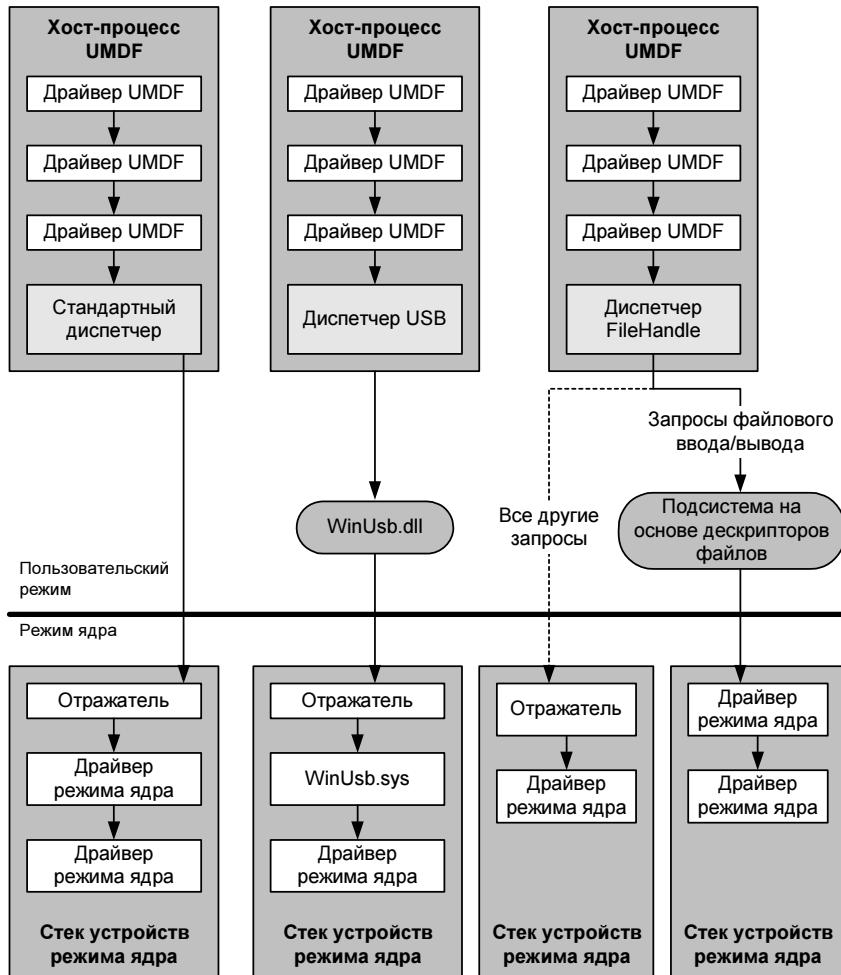


Рис. 9.1. Диспетчеры ввода/вывода UMDF

Как можно видеть на рис. 9.1, диспетчер ввода/вывода предоставляет соединение между хост-процессом UMDF — и, таким образом, частью стека устройств пользовательского режима — и стеком устройств режима ядра. Каждый стек устройств может иметь только одного диспетчера. INF-файл для стека устройств содержит директиву `UmdfDispatcher`, чтобы указать, который из следующих диспетчеров использовать.

- ◆ **Стандартный диспетчер ввода/вывода** отправляет запросы ввода/вывода в стек устройств режима ядра, таким образом фактически объединяя стек пользовательского режима и стек режима ядра в один логический стек устройств. На рис. 9.1 стандартный диспетчер ввода/вывода показан в левом стеке устройств UMDF. Этот диспетчер использует Windows API для отправки запросов ввода/вывода объекту устройства Down отра-

жателя, таким образом вызывая создание нового пакета IRP для стека устройств режима ядра. Стандартный диспетчер ввода/вывода задается в INF-файле, устанавливая значение директивы `UmdfDispatcher` в `Default` или же совсем ее не используя.

- ◆ *Диспетчер USB* отправляет запросы ввода/вывода компоненту пользовательского режима библиотеки WinUSB, который, в свою очередь, отправляет запросы стеку устройств USB режима ядра, где они обрабатываются библиотекой WinUsb.sys. На рис. 9.1 диспетчер USB показан в среднем стеке устройств UMDF. Стеки устройств, использующие получателей ввода/вывода USB, должны пользоваться этим планировщиком. Этот диспетчер задается в INF-файле, присваивая директиве `UmdfDispatcher` значение `WinUsb`.
- ◆ *Диспетчер FileHandle* отправляет запросы ввода/вывода подсистеме или стеку устройств, представленному дескриптором файла. На рис. 9.1 этот диспетчер показан в правом стеке устройств UMDF. Например, если драйвер открывает дескриптор для сокета, канала или другого объекта на основе дескриптора файла, диспетчер `FileHandle` направляет запросы соответствующей подсистеме. Стеки устройств, использующие получатели ввода/вывода `FileHandle`, должны пользоваться этим планировщиком. Этот диспетчер задается в INF-файле, присваивая директиве `UmdfDispatcher` значение `FileHandle`.

На рис. 9.1 показаны два стека устройств режима ядра, ассоциированных со стеком устройств, использующим диспетчер `FileHandle`. Пунктирной линией указывается стандартный узел `devnode`, для которого загружены драйверы UMDF. Запросы, не имеющие дела с файловым вводом/выводом, например, запросы Plug and Play и управления энергопотреблением, следуют по пути, указанному пунктирной линией к стандартному узлу `devnode`. Сплошной линией указывается, каким образом дескриптор файла позволяет драйверу UMDF отправлять запросы ввода/вывода стеку устройств, не являющемуся стандартным узлом `devnode`. Например, драйвер использует получатель ввода/вывода дескриптора файла для обслуживания устройства, подключенного к сети, для которого он открывает сокет для установления связи.

## Внутристековые файлы для получателей ввода/вывода в драйверах UMDF

Дескриптор файла представляет логический сеанс для запросов ввода/вывода. Согласно требованиям UMDF, все запросы ввода/вывода должны иметь сессионную информацию, которая представляется объектом файла и контекстом файла.

При инициализации получателя ввода/вывода, получатели USB и `FileHandle` создают внутристековый файл для представления стандартного сеанса ввода/вывода. Этот сеанс остается открытим до тех пор, пока остается открытим получатель ввода/вывода. Получатели ввода/вывода используют внутристековый файл, чтобы отправлять запросы ввода/вывода, которые не связаны с определенным типом запросов ввода/вывода пользователя. Например, получатель ввода/вывода USB может отправить запрос в этот файл при инициализации, чтобы получить дескриптор конфигурации, или во время обработки, чтобы выбрать интерфейс.

Чтобы получить указатель на интерфейс `IWDFFile` для внутристекового файла, открытого получателем ввода/вывода, драйвер вызывает метод `IWDFIoTarget::GetTargetFile` на объекте получателя ввода/вывода. После этого драйвер может передать указатель интерфейса другим методам интерфейса `IWDFIoTarget` на объекте получателя ввода/вывода, чтобы отправлять запросы ввода/вывода этому файлу.

Используя этот файл, драйвер может отправлять запросы ввода/вывода в том же самом сеансе, что и получатель ввода/вывода. В альтернативе, драйвер может создать собственный сеанс, вызывая метод `IWDFDevice::CreateWdfFile`, таким образом создавая свой внутристековый файл.

Инфраструктура закрывает внутристековый файл, когда драйвер вызывает метод `IWDFObject::DeleteWdfObject` на объекте получателя ввода/вывода или когда инфраструктура удаляет объект устройства, являющийся родителем объекта получателя ввода/вывода.

### **Файловые объекты и внутристековые файлы**

Файловые объекты являются одними из тех объектов ядра, которые не сохранились с изменениями в Plug and Play при переходе от Windows NT4.0 к Windows 2000. В Windows NT 4.0 стеки устройств обычно были неглубокими, и драйверы создавали соединения между устройствами в многоуровневых стеках, открывая лежащее в основе устройство по имени чаще, чем подсоединяясь к стеку устройств. К стеку устройств в основном подсоединялись фильтры. Если фильтру было необходимо хранить посекансное состояние, он мог наложить это состояние на объект файла нижележащего драйвера.

В Windows 2000 стеки устройств стали более многоуровневыми и механизм уровней был перенесен на подключение к устройству. Подключение предоставляет численные преимущества, но т. к. драйверы больше не "открывают" нижележащих драйверов, то между драйверами больше нет никаких файловых объектов.

С приходом UMDF нам снова потребовались внутристековые сеансы, т. к. часть режима ядра стека устройства видит драйвер UMDF просто как еще одного клиента. Управление информацией сеанса является ответственностью в основном объекта FDO и инкапсулируется в файловый объект получателя ввода/вывода.

Принимая во внимание это дополнительное требование для драйверов UMDF, мы тоже решили принести некоторую пользу. Поэтому мы предоставили механизм для поддержки внутристековых сеансов (файловые объекты) в части пользовательского режима стека устройств. (*Питер Виланд (Peter Wieland), команда разработчиков Windows Driver Foundation, Microsoft.*)

## **Создание и управление получателями ввода/вывода**

Для получения доступа к стандартному получателю ввода/вывода драйвер просто вызывает инфраструктурный метод на объекте устройства. Но прежде чем драйвер может использовать удаленный получатель ввода/вывода, он должен создать или открыть объект WDF, представляющий объект получателя ввода/вывода, и ассоциировать этот объект с объектом устройства.

### **Обращение к получателю ввода/вывода**

Инфраструктура создает или открывает стандартный объект получателя ввода/вывода при создании объекта устройства.

Драйвер UMDF может обращаться к стандартному получателю ввода/вывода посредством интерфейса `IWDFIoTarget`; он получает указатель на этот интерфейс, вызывая метод `IWDFDevice::GetDefaultIoTarget`.

Драйвер KMDF получает дескриптор стандартного получателя ввода/вывода, вызывая метод `WdfDeviceGetIoTarget`, передавая ему дескриптор объекта устройства.

## Создание удаленных получателей ввода/вывода в драйверах KMDF

Прежде чем драйвер KMDF может отправлять запросы ввода/вывода удаленному получателю ввода/вывода, он должен создать или открыть получатель ввода/вывода.

Драйвер создает удаленный получатель ввода/вывода, вызывая метод `WdfIoTargetCreate`. Этому методу передаются три параметра: дескриптор объекта устройства, который будет отправлять запросы ввода/вывода получателю, необязательный указатель на структуру `WDF_OBJECT_ATTRIBUTES`, и адрес, по которому инфраструктура возвращает дескриптор созданного объекта получателя ввода/вывода. По умолчанию объект получателя ввода/вывода является потомком указанного объекта устройства.

При создании объекта получателя ввода/вывода, он еще не ассоциирован ни с каким определенным получателем ввода/вывода. Чтобы ассоциировать объект получателя с другим объектом устройства, стеком устройств или файлом, драйвер должен вызвать метод `WdfIoTargetOpen`. В качестве входных параметров методу `WdfIoTargetOpen` передается дескриптор объекта получателя ввода/вывода и указатель на структуру `WDF_IO_TARGET_OPEN_PARAMS`, которая предоставляет многочисленные параметры для получателя ввода/вывода.

Драйвер может позже закрыть получатель ввода/вывода и использовать объект `WDFIOTARGET` с другим получателем, вызывая метод `WdfIoTargetOpen` опять, но с другими значениями в структуре `WDF_IO_TARGET_OPEN_PARAMS`.

### Функции инициализации для структуры параметров получателя ввода/вывода

Структура параметров получателя ввода/вывода предоставляет информацию, требуемую инфраструктурой для открытия получателя ввода/вывода, например, имя устройства получателя и обратные вызовы, которые драйвер реализует для определенных событий получателя ввода/вывода.

Инфраструктура KMDF содержит несколько функций инициализации семейства `WDF_IO_TARGET_OPEN_PARAMS_INIT_XXX`, которые, в зависимости от типа открываемого объекта, заполняют различные поля структуры параметров получателя. В зависимости от того, каким образом драйвер открывает получатель ввода/вывода, применяется одна из функций инициализации, описанных в табл. 9.1.

**Таблица 9.1. Функции инициализации**

Действие драйвера	Применяемая функция
Идентифицирует получателя по имени устройства, интерфейса устройства или файла, и требуется, чтобы инфраструктура создала новый файл	Версия <code>CREATE_BY_NAME</code> функции инициализации, которой передается маска доступа в параметре <code>DesiredAccess</code> , указывающем права доступа, требуемые драйвером
Идентифицирует получателя по имени, и требуется, чтобы инфраструктура завершила неудачей запросы на открытие, если указанный получатель не существует	Версия <code>OPEN_BY_NAME</code> функции инициализации, которой передается маска доступа в параметре <code>DesiredAccess</code> , указывающем права доступа, требуемые драйвером

Таблица 9.1 (окончание)

Действие драйвера	Применяемая функция
Предоставляет указатель на объект DEVICE_OBJECT WDM	Версия EXISTING_DEVICE функции инициализации. Применяя эту функцию инициализации, драйвер, имеющий указатель на объект устройства WDM для другого стека устройств, может использовать этот стек в качестве удаленного получателя ввода/вывода

Функции инициализации заполняют поля в структуре, указывающие, каким образом открывать получатель — по имени или по объекту устройства, имя или указатель на объект устройства, определяющий получателя, и права доступа, требуемые драйвером. Наиболее распространенным способом открытия удаленного получателя ввода/вывода является версия CREATE\_BY\_NAME функции инициализации.

В зависимости от конкретной версии, функции инициализации семейства WDF\_IO\_TARGET\_OPEN\_PARAMS\_INIT\_XXX требуют один или несколько из следующих параметров:

◆ **Имя получателя ввода/вывода или указатель на объект устройства.**

Чтобы идентифицировать получателя ввода/вывода, драйвер предоставляет либо строку в кодировке Unicode, либо указатель на объект устройства.

Строка Unicode представляет имя объекта Windows, которое однозначно указывает устройство, файл, имя символьной ссылки или интерфейс устройства. Если драйвер использует версию OPEN\_BY\_NAME функции инициализации, вызов метода WdfIoTargetOpen завершится неудачей, если объект устройства получателя или интерфейс не существует.

Дополнительную информацию см. в разделе **Object Names** (Имена объектов) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80615>.

◆ **Требуемый доступ.**

WDK определяет несколько констант, описывающих типичные наборы прав доступа. Права доступа применяются на стеках устройств файловой системы, но не имеют значения для многих драйверов. Далее приводятся наиболее употребляемые права доступа для объекта получателя ввода/вывода:

- STANDARD\_RIGHTS\_READ — предоставляет право чтения из получателя;
- STANDARD\_RIGHTS\_WRITE — предоставляет право записи в получатель;
- STANDARD\_RIGHTS\_ALL — предоставляет права доступа к получателю на чтение, запись и удаление.

Полный список прав доступа см. в разделе **ACCESS\_MASK** в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80616>.

Для большинства получателей ввода/вывода функции инициализации семейства WDF\_IO\_TARGET\_OPEN\_PARAMS\_INIT\_XXX заполняют все поля, требуемые драйвером, за исключением указателей для функций обратного вызова по событию. Эти указатели драйвер должен заполнить отдельно. В табл. 9.2 перечислены все поля структуры параметров получателя ввода/вывода.

Если получатель является файлом и драйвер открывает его по имени, драйвер может также заполнить поля, обычно применяемые при создании файла, например, поля FileAttributes и CreateOptions. Но большинство драйверов не предоставляет значений для этих полей, т. к.

**Таблица 9.2. Поля структуры WDF\_IO\_TARGET\_OPEN\_PARAMS**

Имя поля	Описание
Type	Одно из следующих значений перечисления WDF_IO_TARGET_OPEN_TYPE: <ul style="list-style-type: none"><li>• WdfIoTargetOpenUseExistingDevice;</li><li>• WdfIoTargetOpenByName;</li><li>• WdfIoTargetOpenReopen.</li></ul> Заполняется для драйвера функциями инициализации
EvtIoTargetQueryRemove	Указатель на функцию обратного вызова EvtIoTargetQueryRemove
EvtIoTargetRemoveCanceled	Указатель на функцию обратного вызова EvtIoTargetRemoveCanceled
EvtIoTargetRemoveComplete	Указатель на функцию обратного вызова EvtIoTargetRemoveComplete
TargetDeviceObject	Указатель на объект DEVICE_ОБЪЕКТ WDM; заполняется по требованию функцией инициализации; требуется для версии EXISTING_DEVICE функции инициализации
TargetFileObject	Указатель на объект FILE_ОБЪЕКТ; применяется только для типа WdfIoTargetOpenUseExistingDevice
TargetDeviceName	Строка в кодировке Unicode, идентифицирующая получателя ввода/вывода по имени; заполняется по требованию функцией инициализации; требуется для версии OPEN_BY_NAME функции
DesiredAccess	Значение типа ACCESS_MASK; заполняется по требованию функцией инициализации
ShareAccess	Битовая маска, содержащая от нуля до нескольких следующих флагов: FILE_SHARE_READ, FILE_SHARE_WRITE или FILE_SHARE_DELETE
FileAttributes	Битовая маска флагов атрибутов семейства FILE_ATTRIBUTE_Xxxx; значение по умолчанию — FILE_ATTRIBUTE_NORMAL; заполняется по требованию функциями инициализации
CreateDisposition	Константа, определяющая действия системы для создания файла
CreateOptions	Битовая маска флагов опций; заполняется по требованию функциями инициализации
EaBuffer	Буфер расширенных атрибутов для создания файла
EaBufferLength	Размер буфера расширенных атрибутов
AllocationSize	Начальный размер в байтах выделяемый получателю, если получатель является файлом
FileInformation	Возвращаемая информация статуса, если метод WdfIoTargetOpen создает файл

функция инициализации заполняет их необходимыми значениями для драйвера. Эти поля редко применяются, если получатель не является файлом.

Дополнительную информацию об опциях при создании файла см. в разделе **ZwCreateFile** в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80617>.

## Пример KMDF: создание и открытие удаленного получателя ввода/вывода

В листинге 9.1 приведен пример создания и открытия удаленного получателя образцом драйвера Toastmon. Этот драйвер создает по одному получателю ввода/вывода для каждого устройства Toaster в системе. Код примера взят из файла Toastmon.c.

### Листинг 9.1. Создание и открытие удаленного получателя ввода/вывода в драйвере KMDF

```
NTSTATUS Toastmon_OpenDevice(
    WDFDEVICE Device,
    PUNICODE_STRING SymbolicLink,
    WDFIOTARGET *Target)
{
    NTSTATUS status = STATUS_SUCCESS;
    WDF_IO_TARCKET_OPEN_PARAMS openParams;
    WDFIOTARGET ioTarget;
    WDF_OBJECT_ATTRIBUTES attributes;
    // [1]
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes,
                                            TARCET_DEVICE_INFO);
    // [2]
    status = WdfIoTargetCreate(deviceExtension->WdfDevice,
                               &attributes, &ioTarget);
    if (!NT_SUCCESS(status))
    {
        return status;
    }
    . . . // Код для организации таймеров опущен.
    // [3]
    WDF_IO_TARCKET_OPEN_PARAMS_INIT_OPEN_BY_NAME(sopenParams,
                                                SymbolicLink,
                                                STANDARD_RIGHTS_ALL);
    // [4]
    openParams.ShareAccess = FILE_SHARE_WRITE | FILE_SHARE_READ;
    // [5]
    openParams.EvtIoTargetQueryRemove = ToastMon_EvtIoTargetQueryRemove;
    openParams.EvtIoTargetRemoveCanceled =
        ToastMon_EvtIoTargetRemoveCanceled;
    openParams.EvtIoTargetRemoveComplete =
        ToastMon_EvtIoTargetRemoveComplete;
    // [6]
    status = WdfIoTargetOpen(ioTarget, &openParams);
    if (!NT_SUCCESS(status))
    {
        WdfObjectDelete(ioTarget);
        return status;
    }
    . . . // Код опущен.
    return status;
}
```

В листинге 9.1 драйвер создает область контекста для объекта получателя ввода/вывода, создает объект получателя ввода/вывода и открывает получателя по имени. Драйвер также создает таймер, который периодически посыпает запросы ввода/вывода устройству получа-

теля. Код для создания таймера и для отправки запросов ввода/вывода в листинге 9.1 не показан. Далее приводится объяснение пронумерованных фрагментов листинга.

1. Чтобы инициализировать область контекста и атрибуты объекта для объекта получателя ввода/вывода, драйвер вызывает макрос `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE`, передавая ему в качестве параметров указатель на структуру `WDF_OBJECT_ATTRIBUTES` и тип контекста. Тип контекста определен в файле `Toastmon.h`.
2. Драйвер создает объект получателя ввода/вывода, вызывает метод `WdfIoTargetCreate`. Этому методу передаются в качестве параметров дескриптор объекта устройства и указатель на инициализированную структуру атрибутов; метод возвращает дескриптор нового созданного объекта получателя ввода/вывода.
3. Драйвер инициализирует структуру `WDF_IO_TARGET_OPEN_PARAMS` информацией, описывающей, каким образом инфраструктура должна открыть получатель. Требуемая информация вводится функцией `WDF_IO_TARGET_OPEN_PARAMS_INIT_OPEN_BY_NAME`. В качестве параметров драйвер передает этой функции указатель на структуру `WDF_IO_TARGET_OPEN_PARAMS`, имя устройства, которое нужно открыть, и константу `STANDARD_RIGHTS_ALL`, которая указывает права доступа на чтение, запись и удаление.
4. Драйвер устанавливает поле `ShareAccess` структуры `WDF_IO_TARGET_OPEN_PARAMS` и регистрирует функции обратного вызова для объекта получателя ввода/вывода. Поле `ShareAccess` указывает, требует ли драйвер исключительного доступа к получателю. Драйвер позволяет разделяемый доступ на чтение и запись, поэтому полю присваивается объединение значений для этих двух типов доступа.
5. Образец драйвера регистрирует функции обратного вызова для событий `EvtIoTargetQueryRemove`, `EvtIoTargetRemoveCanceled` и `EvtIoTargetRemoveComplete`. Эти обратные вызовы позволяют драйверу выполнять специальную обработку при удалении устройства, непосредственно перед удалением устройства и при отмене удаления. Образец драйвера с помощью таймера отправляет периодические запросы получателю ввода/вывода. Если получатель просто удаляется или удаляется по результатам опроса, драйвер выключает таймер, а если удаление отменяется, драйвер перезапускает таймер. По умолчанию инфраструктура останавливает получатель ввода/вывода при удалении устройства или удалении по результатам опроса и перезапускает его, если удаление отменяется. Для потребностей многих драйверов достаточно обработки, предоставляемой по умолчанию. Драйвер может реализовать обратные вызовы, чтобы заменить предоставляемую по умолчанию обработку своей. В таком случае, инфраструктура не выполняет никаких действий по умолчанию, и весь код для обработки должен быть реализован в драйвере.
6. Драйвер открывает объект, вызывая метод `WdfIoTargetOpen`, передавая ему в качестве параметров дескриптор получателя ввода/вывода и указатель на структуру конфигурации.

## Управление состояниями получателя ввода/вывода

Большим преимуществом получателей ввода/вывода над простыми указателями объектов устройств является возможность драйвера опрашивать получателя о его состоянии и управлять им. Объекты получателей ввода/вывода отслеживают находящиеся в очереди и отправленные запросы, и они могут отменить их в случае изменений в состоянии устройства получателя или в состоянии драйвера WDF, отправившего запрос.

Если при отправлении драйвером запроса получатель ввода/вывода находится в остановленном состоянии, то инфраструктура может поставить запрос в очередь для обработки,

когда получатель возвратится в рабочее состояние. С точки зрения драйвера, объект получателя ввода/вывода ведет себя подобно огражденной от отмен очереди, которая сохраняет отправленные запросы до тех пор, пока инфраструктура не сможет доставить их. Инфраструктура не освобождает объект получателя ввода/вывода до тех пор, пока все отправленные получателю запросы ввода/вывода не будут завершены.

В табл. 9.3 приведен список возможных состояний получателя ввода/вывода.

**Таблица 9.3. Состояния получателя ввода/вывода**

Состояние	Описание
<i>Started</i> (Запущен)	Получатель ввода/вывода запущен и может обрабатывать запросы ввода/вывода
<i>Stopped</i> (Остановлен)	Получатель ввода/вывода остановлен
<i>Stopped for query-remove</i> (Остановлен для удаления по результатам опроса)	Получатель ввода/вывода временно остановлен, т. к. ожидается удаление его устройства. Если, в конечном счете, устройство удалено, то получатель переходит в состояние <i>Deleted</i> . Если устройство не удаляется, то получатель обычно возвращается в состояние <i>Started</i>
<i>Closed</i> (Закрыт)	Удаленный получатель ввода/вывода был закрыт, т. к. драйвер вызвал метод закрытия
<i>Deleted</i> (Удален)	Устройство получателя ввода/вывода было удалено. Инфраструктура отменяет все запросы ввода/вывода, поставленные в очередь для получателя ввода/вывода

По умолчанию инфраструктура отправляет запросы ввода/вывода только тогда, когда получатель ввода/вывода пребывает в состоянии *Started*. Но драйвер также может дать указание инфраструктуре игнорировать состояние получателя и отправить запрос, даже если получатель пребывает в состоянии *Stopped*.

### Методы для управления состоянием получателя ввода/вывода

WDF предоставляет методы, которые драйвер может вызывать для запуска и остановки получателя ввода/вывода, закрытия или удаления его при удалении устройства получателя, и удаления получателя синхронно и опроса его текущего состояния. А именно:

- ◆ драйверы UMDF вызывают методы интерфейса `IWDFIoTargetStateManagement`, реализованного на объекте устройства ввода/вывода;
- ◆ драйверы KMDF вызывают методы семейства `WdfIoTargetXxx`.

В табл. 9.4 приводится сводка методов, вызываемых драйверами для управления состоянием получателя ввода/вывода. KMDF поддерживает дополнительные методы для формирования и отправки запросов получателям ввода/вывода и для опроса свойств удаленных получателей ввода/вывода.

**Таблица 9.4. Методы для управления состоянием получателя ввода/вывода**

Задача	Метод UMDF <code>IWDFIoTargetStateManagement</code>	Метод KMDF
Возвратить текущее состояние получателя ввода/вывода	<code>GetState</code>	<code>WdfIoTargetGetState</code>

Таблица 9.4 (окончание)

Задача	Метод UMDF IWDFIoTargetStateManagement	Метод KMDF
Открыть удаленный получатель ввода/вывода	Никакой	WdfIoTargetOpen
Закрыть удаленный получатель ввода/вывода	Никакой	WdfIoTargetClose
Удалить объект получателя ввода/вывода	Remove	Никакой
Начать отправку запросов объекту получателя ввода/вывода	Start	WdfIoTargetStart
Прекратить отправку запросов ввода/вывода получателю ввода/вывода	Stop	WdfIoTargetStop
Временно закрыть получатель ввода, пока выполняется операция удаления по результатам опроса	Никакой	WdfIoTargetCloseForQueryRemove

### Обратные вызовы получателей ввода/вывода для драйверов KMDF

По умолчанию, если ассоциированное с удаленным получателем ввода/вывода устройство удалено, KMDF останавливает и закрывает объект получателя ввода/вывода, но не извещает драйвер. Однако драйвер может зарегистрировать одну или несколько функций обратного вызова по событию, с тем, чтобы он мог получать извещения, когда удаленный получатель остановлен, остановлен для удаления по результатам опроса или удален.

Если драйвер должен выполнить какую-либо специализированную обработку запросов ввода/вывода, отправленных им получателю ввода/вывода, он должен зарегистрировать одну или несколько функций обратного вызова для событий семейства *EvtIoTargetXxx*, как описано в табл. 9.5. Когда запрашивается, отменяется или завершается удаление устройства, KMDF вызывает соответствующую функцию обратного вызова, после чего обрабатывает изменения состояний получателя ввода/вывода. Драйвер может завершить неудачей запрос на удаление по результатам опроса, реализуя функцию обратного вызова для события *EvtIoTargetQueryRemove* и возвращая статус неудачного завершения.

Таблица 9.5. Функции обратного вызова для событий получателя ввода/вывода

Обратный вызов	Описание
<i>EvtIoTargetQueryRemove</i>	Вызывается, когда необходимо опросить устройство получателя ввода/вывода о возможности его безопасного удаления. Чтобы позволить удаление, функция должна вызвать метод <i>WdfIoTargetCloseForQueryRemove</i> и возвратить статус <i>STATUS_SUCCESS</i> . Чтобы запретить удаление, функция должна возвратить статус неуспеха, например, <i>STATUS_UNSUCCESSFUL</i> или <i>STATUS_INVALID_DEVICE_REQUEST</i> .

Таблица 9.5 (окончание)

Обратный вызов	Описание
EvtIoTargetRemoveCanceled	Вызывается после того, как операция удаления по результатам опроса для устройства получателя была отменена. Эта функция должна снова открыть получатель ввода/вывода, временно остановленный функцией <code>EvtIoTargetQueryRemove</code> , вызывая метод <code>WdfIoTargetOpen</code> со значением <code>WDF_IO_TARGET_OPEN_TYPE</code> . Эта функция должна всегда возвращать статус <code>STATUS_SUCCESS</code>
EvtIoTargetRemoveComplete	Вызывается по завершению удаления устройства получателя ввода/вывода. Эта функция должна всегда возвращать статус <code>STATUS_SUCCESS</code>

Для стандартных получателей ввода/вывода такие функции обратного вызова не определяются. Драйвер WDF и устройство получателя ввода/вывода находятся в том же самом стеке устройств, поэтому драйвер извещается о запросах на удаление устройства через функции обратного вызова для событий Plug and Play и энергопотребления.

При удалении получателя ввода/вывода KMDF по умолчанию отменяет все запросы ввода/вывода, отправленные получателю, и ожидает завершения всех запросов, прежде чем удалять объект устройства ввода/вывода. Если обработка по умолчанию не является достаточной, драйвер должен предоставить специальную процедуру очистки, регистрируя обратный вызов функции `EvtObjectCleanup` для объекта устройства ввода/вывода. Прежде чем инфраструктура выполнит собственную обработку очистки, она активирует обратный вызов очистки. Например, некоторые драйверы могут блокировать новые запросы и ожидать завершения незаконченных запросов ввода/вывода, вместо того чтобы позволить инфраструктуре отменить их. В этом случае, функция обратного вызова `EvtObjectCleanup` вызовет метод `WdfIoTargetStop`, указывая действие `WdfIoTargetWaitForSentIoToComplete`.

### Пример KMDF: функция обратного вызова `EvtIoTargetQueryRemove`

В листинге 9.2 приведен исходный код функции обратного вызова `EvtIoTargetQueryRemove` образца драйвера Toastmon. Инфраструктура вызывает эту функцию в процессе удаления по результатам опроса. В образце драйвера эта функция обратного вызова останавливает таймер и временно закрывает дескриптор объекта получателя ввода/вывода.

#### Листинг 9.2. Функция обратного вызова `EvtIoTargetQueryRemove` в драйвере KMDF

```
NTSTATUS ToastMon_EvtIoTargetQueryRemove (WDFIOTARGETCET IoTarget)
{
    PACED_CODE();
    . . . // Код таймера опущен.
    WdfIoTargetCloseForQueryRemove (IoTarget);
    return STATUS_SUCCESS;
}
```

Чтобы предотвратить удаление получателя, функция `EvtIoTargetQueryRemove` возвращает код ошибки, обычно равный `STATUS_UNSUCCESSFUL` или `STATUS_INVALID_DEVICE_REQUEST`.

Если же функция обратного вызова драйвера `EvtIoTargetQueryRemove` возвращает `STATUS_SUCCESS`, таким образом, разрешая удаление получателя, она должна вызвать метод

WdfIoTargetCloseForQueryRemove. Метод WdfIoTargetCloseForQueryRemove временно закрывает дескриптор получателя ввода/вывода, но не удаляет объект получателя.

Если же устройство в самом деле удаляется, инфраструктура вызывает функцию обратного вызова драйвера EvtIoTargetRemoveComplete, которая очищает все незаконченные запросы, отправленные таймером, и удаляет объект получателя ввода/вывода. Если удаление отменяется, то инфраструктура вызывает функцию обратного вызова EvtIoTargetRemoveCanceled, которая снова открывает получатель и перезапускает таймер. В этом обратном вызове драйвер снова вызывает метод WdfIoTargetOpen, чтобы открыть получателя, но использует функцию WDF\_IO\_TARGET\_OPEN\_PARAMS\_INIT\_REOPEN для инициализации параметров. Эта функция заполняет структуру параметров теми же самыми значениями, которые были использованы драйвером, когда он первоначально открыл получатель ввода/вывода.

## Создание запроса ввода/вывода

Созданный драйвером объект запроса ввода/вывода описывает запрос на чтение, запись или IOCTL. Драйвер может создавать собственные объекты запроса ввода/вывода, чтобы послать новый запрос ввода/вывода своему стеку устройств, запросить ввод/вывод у другого устройства или для того, чтобы разбить запрос ввода/вывода на несколько меньших запросов перед тем, как завершить его.

Когда драйвер вызывает инфраструктуру для создания запроса ввода/вывода, инфраструктура создает и инициализирует объект запроса ввода/вывода WDF. Объект запроса не содержит объекта памяти или какой-либо другой информации; все это поставляется драйвером позже, при форматировании запроса.

Кроме отправки запросов на чтение, запись и IOCTL, драйверы KMDF могут отправлять внутренние запросы IOCTL. Эти запросы определяются устройством, и в них иногда применяются нестандартные параметры. Вместо предоставления входных и выходных буферов для таких запросов драйвер предоставляет до трех объектов памяти и структур смещений, которые описывают буферное пространство для запроса.

### Пример UMDF: создание объекта запроса ввода/вывода WDF

Для создания объекта запроса ввода/вывода драйвер UMDF вызывает метод IWDFDevice::CreateRequest. В качестве параметров методу передаются указатель на интерфейс IUnknown на объекте обратного вызова запроса драйвера, указатель на интерфейс IWDFObject на родительском объекте и адрес для получения указателя на интерфейс IWDFIoRequest на созданном объекте запроса.

В листинге 9.3 приведен пример создания запроса, взятый из файла Device.cpp образца драйвера Fx2\_Driver.

#### Листинг 9.3. Создание запроса ввода/вывода WDF в драйвере UMDF

```
HRESULT hr = S_OK;
IWDFIoRequest *pWdfRequest = NULL;
IWDFDriver * FxDriver = NULL;
hr = m_FxDevice->CreateRequest( NULL, // pCallbackInterface,
                                NULL, // pParentObject
                                &pWdfRequest);
```

Как можно видеть, создание объекта запроса ввода/вывода является довольно прямолинейной задачей. Драйвер вызывает метод `CreateRequest` объекта устройства, передавая ему в параметрах три указателя:

- ◆ указатель на интерфейс обратного вызова, используемый инфраструктурой для запроса функции обратного вызова очистки объекта запроса. Образец драйвера передает в этом параметре значение `NULL`, т. к. он не выполняет никакой очистки, связанной с запросом;
- ◆ указатель на родительский объект. Образец драйвера передает значение `NULL`, указывая принять объект устройства в качестве родителя по умолчанию;
- ◆ указатель на переменную, в которой инфраструктура возвращает указатель на интерфейс `IWDFIoRequest` на созданном объекте запроса.

## **Пример KMDF: создание объекта запроса ввода/вывода WDF**

Для создания объекта запроса ввода/вывода WDF драйвер KMDF вызывает метод `WdfRequestCreate`. В качестве параметров этому методу передается указатель на структуру атрибутов объекта, необязательный дескриптор объекта получателя ввода/вывода, и адрес для получения дескриптора созданного объекта `WDFREQUEST`.

В листинге 9.4 приведен пример создания запроса ввода/вывода, взятый из файла `Toastmon.c` образца драйвера `Toastmon`.

### **Листинг 9.4. Создание запроса ввода/вывода WDF в драйвере KMDF**

```
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.ParentObject = ioTarget;
status = WdfRequestCreate(&attributes, ioTarget,
                         &targetDeviceInfo->ReadRequest);
```

Как можно видеть в листинге 9.4, драйвер инициализирует структуру атрибутов, после чего присваивает родительскому объекту ранее созданный объект получателя ввода/вывода. Установливая объект получателя ввода/вывода в качестве родителя, драйвер обеспечивает тот факт, что инфраструктура удалит объект запроса ввода/вывода при удалении объекта получателя ввода/вывода. В этом случае драйвер продолжает использовать для запроса тот же самый получатель ввода/вывода до тех пор, пока он является доступным.

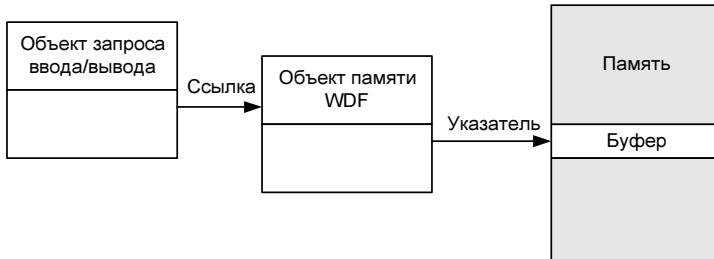
Потом драйвер создает запрос, вызывая для этого метод `WdfRequestCreate`. В качестве параметров методу передаются указатель на структуру атрибутов, дескриптор объекта получателя ввода/вывода и адрес, по которому инфраструктура возвращает дескриптор созданного объекта запроса ввода/вывода. В образце драйвера этот дескриптор сохраняется в области контекста объекта получателя ввода/вывода.

## **Объекты памяти и буферы для созданных драйвером запросов ввода/вывода**

Любой запрос ввода/вывода, созданный драйвером, требует один или несколько объектов памяти, описывающих буферы ввода/вывода для запроса. Прежде чем отправить запрос ввода/вывода, драйвер должен предоставить объекты памяти и буферы для запроса и отформатировать запрос.

Драйверы, обрабатывающие запрос ввода/вывода, выполняют чтение и запись в буферы, поставляемые с запросом. Для запросов на чтение буфер вывода предоставляет адрес, по которому получатель ввода/вывода возвращает запрошенные данные. Для запросов на чтение буфер вывода предоставляет данные, которые нужно записать. Запрос IOCTL может иметь один буфер для ввода и другой буфер для вывода или использовать один и тот же буфер как для ввода, так и для вывода.

Взаимоотношения между объектом запроса ввода/вывода WDF, объектом памяти WDF и буфером показаны на рис. 9.2.



**Рис. 9.2.** Взаимоотношения между объектом запроса ввода/вывода WDF, объектом памяти WDF и буфером

Как можно видеть на рис. 9.2, объект запроса ввода/вывода WDF содержит ссылку на объект памяти. Объект памяти, в свою очередь, содержит указатель на буфер. С некоторыми ограничениями, WDF предоставляет методы для установки и извлечения объекта памяти из объекта запроса ввода/вывода и для установки и извлечения указателя на буфер из объекта памяти.

Объекты памяти являются объектами WDF, поэтому они не просто служат оберткой для указателей буферов, но также содержат счетчики ссылок. Применение счетчиков ссылок для памяти уменьшает шансы, что драйвер ненароком освободит память в неподходящий момент или попытается обратиться к памяти, которая уже была освобождена. Применение счетчиков ссылок предоставляет следующие преимущества.

- ◆ Память нельзя освободить до тех пор, пока не будет завершен запрос ввода/вывода WDF, использующий ее, т. к. объект запроса ввода/вывода удерживает ссылку на объект памяти. Эта ссылка также обеспечивает, что память не будет освобождена, пока запрос ожидает исполнения в драйвере получателя ввода/вывода.
- ◆ Это особенно полезно при отладке, т. к. такая ошибка вызывает останов bugcheck, что указывает на проблему в драйвере получателя, хотя в действительности ошибка находится в вашем драйвере.
- ◆ Упрощается очистка ресурсов в драйвере, отправляющем запросы, т. к. драйвер может удалить объект памяти, пока запрос ввода/вывода WDF все еще ожидает выполнения. Объект памяти остается достоверным до тех пор, пока запрос ввода/вывода не завершен.

Хотя для объекта памяти имеется счетчик ссылок на него, такого счетчика нет для лежащего в его основе буфера. Буфер представляет собой лишь область памяти, выделенную приложением, инфраструктурой или драйвером. Это не объект WDF, поэтому его время жизни управляет компонентом, создавшим его. Если драйвер отправляет запросы ввода/вывода, и особенно если он многократно использует объекты памяти, необходимо быть внимательным, каким образом и когда драйвер обращается к указателю на буфер. Только ненулевое

значение счетчика ссылок само по себе не является достаточным для удержания указателя на буфер достоверным. Сводка времени жизни буфера приведена в табл. 9.6.

**Таблица 9.6. Время жизни буфера**

<b>Если буфер был создан...</b>	<b>Указатель на буфер становится недействительным, когда...</b>
Приложением, выдавшим запрос ввода/вывода	Драйвер завершает запрос ввода/вывода
Вызванной драйвером языковой или системной функцией выделения памяти	Драйвер освобождает выделенную память
Инфраструктурным методом IWDFDriver::CreateWdfMemory или WdfMemoryCreate	Счетчик ссылок на объект памяти примет нулевое значение

#### **Управление объектами памяти, имеющими разное время жизни**

Необходимо быть осторожным при совместном использовании в коде объектов памяти, имеющих разное время жизни буферов. Чтобы упростить написание кода, старайтесь использовать только один тип комбинации объекта памяти с буфером в определенном пути ввода/вывода. (Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

### **Выделение объектов памяти и буферов для запросов ввода/вывода**

Драйвер может выделить буфер и объект памяти следующими способами:

- ◆ выделить буфер и объект памяти одновременно, в одном вызове инфраструктуры;
- ◆ создать новый объект памяти и ассоциировать его с существующим буфером, выделенным драйвером.

Драйвер может создать новый объект памяти WDF при создании запроса ввода/вывода, или же он может использовать уже существующий объект памяти, который он создал раньше или извлек из входящего запроса ввода/вывода.

#### **Родитель объекта памяти**

При создании драйвером нового объекта памяти инфраструктура по умолчанию устанавливает объект драйвера в качестве родителя этого объекта памяти. Это стандартное действие предназначается для выделения памяти общего назначения, но является не совсем идеальным для памяти, используемой для запросов ввода/вывода. Если драйвер создает новый объект памяти для использования в запросе ввода/вывода, его родителем нужно назначить объект, чье время жизни близко совпадает с его собственным (табл. 9.7).

**Таблица 9.7. Взаимодействие объекта и его родителя**

<b>Для того, чтобы...</b>	<b>Установите родителем этот объект...</b>
Удалить объект памяти по завершению запроса ввода/вывода	Объект запроса ввода/вывода
Сохранить объект памяти для дальнейшего использования в других запросах ввода/вывода	Объект устройства

Для установки родителя объекта памяти:

- ◆ драйвер UMDF предоставляет указатель на интерфейс `IWDFObject` родительского объекта при создании объекта памяти;
- ◆ драйвер KMDF предоставляет значение для поля `ParentObject` структуры атрибутов объекта, которую он предоставляет при создании объекта памяти.

Если драйвер использует объект памяти, извлеченный им из объекта входящего запроса WDF, этот объект запроса ввода/вывода и является родителем данного объекта памяти. Таким образом, объект памяти не может продолжать существовать больше, чем объект входящего запроса ввода/вывода WDF. По завершению входящего запроса ввода/вывода всеми драйверами, инфраструктура удаляет как объект запроса, так и объект памяти.

### Совет

Если объект памяти, созданный драйвером, которому не был назначен соответствующий родитель, не удалить явным образом, то он может продолжать существовать до тех пор, пока не будет ликвидирован объект драйвера. Если драйвер использует несколько объектов памяти, то это может вызвать использование больших объемов памяти и, возможно, понизить производительность или вызвать неудачное выполнение дальнейших операций выделения памяти. Более того, эта утечка памяти может быть пропущена утилитами для обнаружения утечки памяти, т. к. объекты памяти, в конечном счете, освобождаются.

## Типы буферов

Драйверы UMDF и KMDF используют разные типы буферов, т. к. они обращаются к разным пулам памяти и адресным пространствам.

- ◆ Драйверы UMDF могут использовать только адресное пространство хост-процесса, в котором вся память является страничного типа. Драйверам UMDF не требуется нестраничная память, т. к. весь их код исполняется на уровне `PASSIVE_LEVEL`, поэтому подкакча всегда разрешена. Диспетчер ввода/вывода Windows блокирует все буферы, которые драйвер UMDF передает части стека устройств, работающей в режиме ядра.
- ◆ Драйверы KMDF могут выделять память из системного страничного или нестраничного пула. В синхронных запросах время жизни объекта памяти и буфера одинаковые. Поэтому драйвер может предоставить простой указатель на память ( `PVOID`) или указатель на список MDL ( `PDMList`) для буфера, или же драйвер может сформировать буфер одновременно с созданием объекта памяти. В асинхронном запросе драйвер должен использовать объекты WDF, чтобы инфраструктура могла обеспечить ситуацию, когда буферы продолжают существовать до тех пор, пока процесс завершения запроса ввода/вывода не дойдет до драйвера, создавшего этот запрос.

## Одновременное создание объекта памяти и буфера

Драйвер WDF может создать объект памяти и выделить буфер одним вызовом инфраструктуры с помощью одного из следующих методов:

- ◆ `IWDFDriver::CreateWdfMemory` — создает объект памяти и выделяет буфер указанного размера для драйвера UMDF;
- ◆ `WdfMemoryCreate` — создает объект памяти и выделяет буфер указанного размера для драйвера KMDF.

Время жизни объекта памяти и буфера, созданных инфраструктурой одновременно, одинаково. Инфраструктура обеспечивает, что буфер продолжает существовать до тех пор, пока процесс завершения запроса ввода/вывода не дойдет обратно к драйверу.

Этот метод является самым простым в использовании и требует минимума внимания к управлению временем жизни объекта. Большинство драйверов должно создавать объекты памяти и буферы таким образом всегда, когда это практически.

## **Создание объекта памяти, использующего уже существующий буфер**

Использование драйвером уже существующего буфера может иногда помочь избежать двойной буферизации, т. е. копирования данных из внутреннего буфера драйвера в объект памяти WDF и наоборот. Например, драйвер может уже иметь участок в своей области контекста, содержащий данные, которые нужно послать в запросе IOCTL. Вместо того чтобы выделять новый буфер вместе с новым объектом памяти и копировать данные в новый буфер, драйвер может ассоциировать уже существующий буфер с новым объектом памяти.

Следующие методы создают новый объект памяти, использующий уже существующий буфер:

- ◆ `IWDFDriver::CreatePreallocatedWdfMemory` — создает новый инфраструктурный объект памяти и ассоциирует его с существующим буфером для драйвера UMDF. Драйвер UMDF может позже назначить объекту памяти другой буфер, вызывая метод `IWDFMemory::SetBuffer` объекта памяти;
- ◆ `WdfMemoryCreatePreallocated` — создает новый объект памяти и ассоциирует его с уже существующим буфером для драйвера KMDF. Драйвер KMDF может позже назначить объекту памяти другой буфер, вызывая метод `WdfMemoryAssignBuffer`;
- ◆ `WdfMemoryCreateFromLookaside` — создает новый объект памяти и назначает ему буфер со списка резервных буферов (lookaside list, пул буферов фиксированного размера). Дополнительная информация о списках резервных буферов приводится в главе 12.

Когда драйвер использует уже существующий буфер в объекте памяти WDF, драйвер должен обеспечить, что буфер продолжает существовать до тех пор, пока процесс завершения запроса не вернется обратно к драйверу. Инфраструктура не освобождает буфер, когда объект памяти удаляется. Более того, инфраструктура не освобождает ранее назначенный буфер, когда драйвер назначает объекту памяти новый буфер.

### **Внимание!**

Вот список правил для повторного использования объектов запроса ввода/вывода:

- драйвер может переназначить созданный им объект памяти;
- драйвер может переназначить объект памяти, полученный им от инфраструктуры в запросе ввода/вывода, если ему требуется разбить входящий запрос на меньшие части;
- драйвер может назначить один и тот же объект памяти для одновременного использования несколькими запросами ввода/вывода, если все эти запросы обращаются к буферу только для чтения;
- драйверы UMDF не могут повторно использовать объекты запроса ввода/вывода;
- драйверы KMDF могут повторно использовать созданный ими объект запроса ввода/вывода. Драйвер KMDF не может повторно использовать объект запроса ввода/вывода, полученный им от инфраструктуры.

## **Сопоставление объекта памяти объекту запроса ввода/вывода**

После создания драйвером объекта памяти он должен вызвать инфраструктуру для того, чтобы сопоставить (или ассоциировать) этот объект памяти объекту запроса ввода/вывода и

отформатировать запрос для получателя ввода/вывода. В ответ инфраструктура подготавливает нижележащий в стеке пакет IRP и получает ссылку на объект памяти от имени получателя ввода/вывода. Эта ссылка продолжает существовать до тех пор, пока не произойдет одно из следующих событий:

- ◆ процесс завершения запроса ввода/вывода дойдет обратно до драйвера;
  - ◆ драйвер переформатирует объект запроса ввода/вывода WDF;
  - ◆ объект запроса ввода/вывода WDF будет удален;
  - ◆ при подготовке для отправки объекта запроса ввода/вывода WDF другому получателю ввода/вывода драйвер KMDF вызовет метод `WdfRequestReuse`.

## Подсказка

В главе 24 описывается, каким образом комментировать функции обратного вызова драйвера для того, чтобы SDV мог анализировать их на соответствие правилам KMDF, требующим, чтобы после завершения запроса его буфер, список MDL или объект памяти становились недоступными.

## Пример UMDF: создание объекта памяти, использующего уже существующий буфер

В листинге 9.5 показан пример создания драйвером UMDF нового объекта памяти, использующего уже существующий буфер. Исходный код примера взят из файла Device.cpp образца драйвера Fx2\_Driver.

#### Листинг 9.5. Вызов метода CreatePreallocatedWdfMemory в драйвере UMDF

```

. . . // Код обработки ошибок и вспомогательный код опущены.
}
return;
}

```

Код в листинге 9.5 создает инфраструктурный объект запроса ввода/вывода и инфраструктурный объект памяти, который драйвер отправляет в запросе. Для создания объекта памяти и сопоставления его буферу, полученному функцией `SendControlTransferSynchronously` в параметре от вызвавшего ее клиента, драйвер вызывает метод `CreatePreallocatedWdfMemory` объекта драйвера.

Методу `CreatePreallocatedWdfMemory` передается четыре параметра: указатель на буфер, размер буфера, указатель на интерфейс `IUnknown`, который инфраструктура может опросить на наличие обратных вызовов для очистки объекта, и указатель на интерфейс родительского объекта.

Приведенный в листинге 9.5 пример не реализует объекта обратного вызова для объекта памяти, поэтому в параметре для интерфейса `IUnknown` он передает значение `NULL`. Передавая значение `pWdfRequest` в четвертом параметре, драйвер устанавливает объект запроса ввода/вывода родителем созданного объекта памяти. Но ни объект памяти, ни объект запроса ввода/вывода не является родителем буфера. Так как происхождение буфера зависит от клиента, вызывающего функцию `SendControlTransferSynchronously`, эта функция не может делать предположений о собственнике буфера или о его времени жизни.

После возвращения управления метод `CreatePreallocatedWdfMemory` предоставляет указатель на интерфейс `IWdfMemory` для инфраструктурного объекта памяти.

## Пример KMDF: создание объекта памяти и нового буфера

В листинге 9.6 показан пример создания нового объекта памяти и нового буфера в драйвере KMDF. Исходный код этого примера основан на коде из файла `Sys\Bulkwr.c` образца драйвера `Usbsamp`.

### Листинг 9.6. Создание объекта памяти в драйвере KMDF

```

WDF_OBJECT_ATTRIBUTES          objectAttribs;
WDFREQUEST                   Request;
WDFMEMORY                     urbMemory;
PURE                          urb = NULL;
WDF_OBJECT_ATTRIBUTES_INIT(&objectAttribs);
objectAttribs.ParentObject = Request;
status = WdfMemoryCreate(SobjectAttribs,
                        NonPagedPool,
                        POOL_TAC,
                        sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER),
                        &urbMemory,
                        (PVOID*) &urb);

```

Образец драйвера `Usbsamp` создает новый объект памяти для использования в запросе ввода/вывода, который он отправляет получателю ввода/вывода USB. Он инициализирует структуру `WDF_OBJECT_ATTRIBUTES` для объекта памяти, после чего присваивает полю

ParentObject структуры атрибутов значение объекта запроса ввода/вывода, который будет использовать данный объект памяти. После этого драйвер создает объект памяти, вызывая метод `WdfMemoryCreate`. Этому методу передается четыре входных параметра: указатель на структуру атрибутов объекта, константа перечисления, указывающая тип памяти, которую нужно выделить, тег пула, идентифицирующий драйвер и тип буфера, и размер буфера, который нужно выделить. Метод возвращает дескриптор созданного объекта памяти и указатель на новый выделенный буфер, ассоциированный с этим объектом памяти.

## Форматирование запросов ввода/вывода

Прежде чем драйвер может отправить запрос ввода/вывода, этот запрос должен быть отформатирован. Форматирование подготавливает нижележащий в стеке пакет IRP для доставки получателю ввода/вывода, устанавливая расположения стеков ввода/вывода и предостав员я любую другую требуемую информацию, например, функцию обратного вызова для завершения запроса ввода/вывода.

### Когда нужно форматировать запрос

Драйвер должен форматировать запрос в следующих ситуациях.

- ◆ Если драйвер изменяет формат запроса, полученного им от инфраструктуры, и потом отправляет этот запрос получателю ввода/вывода.  
Под "изменением формата запроса" имеются в виду любые модификации параметров, полученных драйвером в запросе. Распространенным примером таких модификаций является изменение смещения буфера или модификация его размера или содержимого.
- ◆ Если драйвер отправляет созданный драйвером запрос ввода/вывода получателю ввода/вывода.
- ◆ Если драйвер регистрирует обратный вызов завершения для запроса ввода/вывода.

В сущности, драйвер должен форматировать все запросы, за исключением запросов, которые он получает от инфраструктуры и впоследствии отправляет стандартному получателю ввода/вывода без изменений и без обратного вызова для завершения запроса ввода/вывода. Способ отправления таких запросов описывается в разд. *"Опция "отправил и забыл"*" далее в этой главе.

### Форматирование запроса

Драйвер форматирует запрос, вызывая методы WDF для форматирования. Инфраструктура предоставляет методы форматирования, которые:

- ◆ форматируют неизмененный запрос для стандартного получателя ввода/вывода;
- ◆ форматируют созданный драйвером запрос или переформатируют запрос, присланный инфраструктурой;
- ◆ форматируют и отправляют синхронные запросы (только в KMDF).

Если драйвер форматирует запрос ввода/вывода, он также должен зарегистрировать для него обратный вызов для завершения.

## Форматирование неизмененного запроса для стандартного получателя ввода/вывода

Если драйвер регистрирует обратный вызов для завершения ввода/вывода, он должен отформатировать запрос, прежде чем отправлять его стандартному получателю ввода/вывода, даже если драйвер не модифицирует никаких параметров в запросе. Драйвер форматирует такой запрос с помощью метода, реализованного на объекте запроса WDF:

- ◆ драйверы UMDF вызывают метод `IWDFRequest::FormatUsingCurrentType`;
- ◆ драйверы KMDF вызывают метод `WdfRequestFormatRequestUsingCurrentType`.

Эти два метода принимают существующий объект запроса ввода/вывода WDF и организовывают ячейку стека ввода/вывода в нижележащем пакете IRP для следующего нижележащего драйвера, не модифицируя пакет IRP никаким другим образом. Эти методы являются эквивалентом процедуры `IoCopyCurrentIrpStackLocationToNext`, используемой драйверами WDM при отправлении пакетов IRP.

Если по завершению запроса драйверу не требуется обратный вызов, то драйвер не обязан форматировать запрос.

## Форматирование измененных или созданных драйвером запросов

Если драйвер создает объект запроса ввода/вывода или изменяет формат объекта запроса ввода/вывода, полученного от инфраструктуры, драйвер должен отформатировать этот запрос, применяя метод форматирования объекта получателя ввода/вывода.

### Методы форматирования UMDF для запросов ввода/вывода

Драйверы UMDF вызывают следующие методы форматирования на инфраструктурном объекте получателя ввода/вывода:

- ◆ `IWDFIoTarget::FormatRequestForIoctl` — форматирует запрос IOCTL для любого получателя ввода/вывода;
- ◆ `IWDFIoTarget::FormatRequestForRead` — форматирует запрос на чтение для любого получателя ввода/вывода;
- ◆ `IWDFIoTarget::FormatRequestForWrite` — форматирует запрос на запись для любого получателя ввода/вывода.

### Методы форматирования KMDF для запросов ввода/вывода

Драйверы KMDF используют следующие методы форматирования:

- ◆ `WdfIoTargetFormatRequestForInternalIoctl` — форматирует внутренний запрос IOCTL для любого получателя ввода/вывода;
- ◆ `WdfIoTargetFormatRequestForInternalIoctlOthers` — форматирует внутренний запрос IOCTL, требующий нестандартные параметры, для любого получателя ввода/вывода;
- ◆ `WdfIoTargetFormatRequestForIoctl` — форматирует запрос IOCTL для любого получателя ввода/вывода;
- ◆ `WdfIoTargetFormatRequestForRead` — форматирует запрос на чтение для любого получателя ввода/вывода;

- ◆ `WdfIoTargetFormatRequestForWrite` — форматирует запрос на запись для любого запроса ввода/вывода.

KMDF также предоставляет методы для форматирования и отправки синхронных запросов ввода/вывода в одной посылке. Дополнительную информацию по этому вопросу см. в разд. "Отправление запросов ввода/вывода" далее в этой главе.

## Параметры для методов форматирования

Методы форматирования KMDF и UMDF, реализованные на объектах получателей ввода/вывода, принимают один или несколько параметров, в зависимости от типа запроса:

- ◆ Request — идентифицирует объект запроса ввода/вывода, который нужно форматировать. Требуется для всех запросов;
  - ◆ IoctlCode — указывает код IOCTL. Применяется только для запросов IOCTL;
  - ◆ InputMemory — идентифицирует объект памяти WDF, предоставляющий буфер ввода. Используется для запросов на запись и IOCTL;
  - ◆ InputMemoryOffset — указывает на структуру `WDFMEMORY_OFFSET`, которая предоставляет смещение в буфер ввода и размер буфера. Используется для запросов на запись и IOCTL;
  - ◆ OutputMemory — идентифицирует объект памяти WDF, предоставляющий буфер вывода. Используется для запросов на чтение и IOCTL;
  - ◆ OutputMemoryOffset — указывает на структуру `WDFMEMORY_OFFSET`, которая предоставляет смещение в буфер вывода и размер буфера. Используется для запросов на чтение и IOCTL;
  - ◆ DeviceOffset — указывает смещение в устройстве, с которого нужно начинать передачу. Используется только для запросов на чтение и запись.

Для методов KMDF также требуется предоставить дескриптор получателя ввода/вывода, которому будет отправлен запрос. Для методов UMDF эта информация не требуется, т. к. они реализованы на объекте получателя ввода/вывода.

## Пример UMDF: форматирование запроса на запись

В листинге 9.7 показан пример форматирования драйвером UMDF запроса на запись. Этот пример основан на коде из файла Queue.cpp для образца драйвера Usb\Echo\_driver.

### Листинг 9.7. Форматирование запроса на запись драйвером UMDF

```

        NULL, // Memory offset
        NULL // DeviceOffset );
if (FAILED(hr)) {
    pWdfRequest->Complete(hr);
} else {
    ForwardFormattedRequest(pWdfRequest, pOutputPipe);
}
SAFE_RELEASE(pInputMemory);
return;
}

```

В предыдущем листинге показан метод `IQueueCallbackWrite::OnWrite` очереди ввода драйвера `Echo_driver`. При вызове этого метода инфраструктура передает ему в качестве параметров указатели на интерфейс `IWDFQueue` объекта очереди и интерфейс `IWDFIoRequest` объекта запроса. Драйвер извлекает объект памяти WDF из входящего объекта запроса и передает его методу `FormatRequestForWrite` канала получателя USB для форматирования запроса на чтение для получателя USB.

### Пример KMDF: форматирование запроса на чтение

В листинге 9.8 показан пример форматирования драйвером KMDF запроса на чтение. Этот пример основан на коде из файла `Toastmon.c` для образца драйвера `Toastmon`.

#### Листинг 9.8. Форматирование запроса на чтение драйвером KMDF

```

status = WdfMemoryCreate(&attributes,
                        NonPagedPool,
                        DRIVER_TAC,
                        READ_BUF_SIZE,
                        &memory,
                        NULL); // buffer pointer
if (!NT_SUCCESS(status)) {
    return status;
}
status = WdfIoTargetFormatRequestForRead(IoTarget,
                                         request,
                                         memory,
                                         NULL, // Output buffer offset
                                         NULL); // Device offset

```

К тому времени, когда исполняется код в листинге 9.8, образец драйвера уже создал как объект запроса ввода/вывода, так и объект получателя ввода/вывода и инициализировал структуру атрибутов. В листинге можно видеть, как драйвер создает объект памяти WDF для выходного буфера запроса на чтение, после чего форматирует запрос под использование этого объекта.

Сначала драйвер создает объект памяти WDF, вызывая метод `WdfMemoryCreate`. Этот объект памяти использует буфер, выделенный из нестраничного пула, размер которого в байтах указан в параметре `READ_BUF_SIZE`. Для форматирования запроса драйвер вызывает метод `WdfIoTargetFormatRequestForRead`, передавая ему в параметрах дескрипторы объекта получателя ввода/вывода и объекта запроса ввода/вывода и объект памяти. Для параметров смещения буфера ввода и смещения устройства указываются значения `NULL`.

## Обратные вызовы для завершения ввода/вывода

По умолчанию WDF отправляет запросы ввода/вывода асинхронно. Обычно драйвер регистрирует обратный вызов для завершения ввода/вывода, чтобы получить извещение о завершении асинхронного запроса. Инфраструктура активирует этот обратный вызов после исполнения обратных вызовов для завершения запроса ввода/вывода всех нижележащих драйверов в стеке.

Драйвер должен зарегистрировать обратный вызов завершения ввода/вывода для каждого асинхронного запроса, если только драйвер не установит флаг "отправил и забыл", как описано в разд. *"Отправил и забыл"* далее в этой главе.

- ◆ Драйвер UMDF вызывает метод `IWDFIoRequest::SetCompletionCallback`, передавая ему в параметрах указатель на интерфейс `IRquestCallbackRequestCompletion` на объекте обратного вызова запроса и указатель на определенную драйвером область контекста.
- ◆ Драйвер KMDF вызывает метод `WdfRequestSetCompletionRoutine`, передавая ему указатель на функцию обратного вызова по событию `CompletionRoutine` и указатель на определенную драйвером область контекста.

### Обработка в функции обратного вызова завершения ввода/вывода

В функции обратного вызова для завершения ввода/вывода драйвер выполняет любую дополнительную обработку, требующуюся для завершения запроса. Такая обработка обычно состоит из проверки статуса завершения запроса и извлечения драйвером требующихся ему данных из буферов ввода/вывода запроса.

Если запрос был создан драйвером, то драйвер не должен завершать запрос в функции обратного вызова завершения запроса. Процесс обработки запроса уже и так прошел весь путь обратно к драйверу.

Для запросов, полученных от инфраструктуры и потом отправленных получателю ввода/вывода, драйвер должен вызвать инфраструктуру, чтобы завершить запрос или в функции обратного вызова завершения ввода/вывода и некоторое время спустя. Легче всего представить себе этот процесс, держа в уме, что обработка запросов начинается и оканчивается на одном и том же уровне стека устройств. Если драйвер получает запрос, он может его завершить, т. к. запрос прибыл из высшего уровня. Драйвер не должен никогда завершать созданный им же запрос, т. к. этот запрос не прибыл из высшего уровня.

Если драйвер больше не использует созданный им объект запроса ввода/вывода и никакие из его дочерних объектов памяти, драйвер может удалить этот объект. Но, устанавливая соответствующего родителя объекта запроса, драйвер может избежать необходимости удалять его явным образом.

Вместо удаления созданных им объектов запроса ввода/вывода, драйвер KMDF может повторно использовать их. После того как драйвер KMDF извлечет из запроса все информацию о статусе и результаты ввода/вывода, он может подготовить объект запроса для повторного использования. Подробности этой процедуры описаны в разд. *"Пример KMDF: повторное использование объекта ввода/вывода"* далее в этой главе.

Обратные вызовы как для драйверов UMDF, так и для драйверов KMDF принимают следующие четыре параметра:

- ◆ `Request` — значение для UMDF: указатель на интерфейс `IWDFIoRequest` для завершенного запроса ввода/вывода. Значение для KMDF: дескриптор завершенного объекта `WDFREQUEST`.

- ◆ Target — значение для UMDF: указатель на интерфейс `IWDFIoTarget` для получателя ввода/вывода. Значение для KMDF: дескриптор объекта `WDFIOTARGET`.
- ◆ Params — значение для UMDF: указатель на интерфейс `IWDFRequestCompletionParams` объекта параметров для завершения объекта запроса ввода/вывода. Значение для KMDF: указатель на структуру `WDF_REQUEST_COMPLETION_PARAMS`, содержащую параметры для завершения запроса.
- ◆ Context — для UMDF и KMDF: указатель на определенную драйвером область контекста, предоставленную драйвером при регистрации им обратного вызова.

Параметр `Context` может содержать указатель на любую информацию, которая может требоваться драйверу при завершении им запроса. Возможность использования параметра `Context` вместо области контекста объекта запроса ввода/вывода следует рассмотреть, когда время жизни информации отличается от времени жизни объекта запроса ввода/вывода или когда информация используется совместно запросом и другим объектом.

### **Когда завершать запрос в драйвере WDF**

Если вы знакомы с драйверами WDM, то требование WDF завершить доставленные инфраструктурой запросы ввода/вывода в функции обратного вызова завершения ввода/вывода, возможно, удивит вас.

Но помните, что инфраструктура устанавливает собственную процедуру завершения, которая получает обратно владение пакетом IRP от имени драйвера, что является эквивалентом возвращения значения статуса `STATUS_MORE_PROCESSING_REQUIRED` в драйвере WDM. Драйвер WDF вызывает инфраструктурный метод завершения запроса, чтобы указать, что он закончил обработку запроса WDF с тем, чтобы инфраструктура могла продолжить завершение пакета IRP.

### **Извлечение статуса завершения и другой информации**

Для получения статуса завершения и количества переданных байтов драйвер применяет информацию параметра `Params` следующим образом.

- ◆ Драйвер UMDF вызывает методы интерфейса `IWDFRequestCompletionParams` на объекте запроса ввода/вывода.

Инфраструктура передает указатель на этот интерфейс в качестве входного параметра функции обратного вызова для завершения ввода/вывода. Драйвер может запросить у этого интерфейса указатель на интерфейс `IWDFIoRequestCompletionParams`, который поддерживает методы, возвращающие буферы для запросов на чтение, запись и IOCTL. Эти два интерфейса организованы иерархически, поэтому интерфейс `IWDFIoRequestCompletionParams` является производным интерфейса `IWDFRequestCompletionParams`.

- ◆ Драйвер KMDF использует указатель на структуру `WDF_REQUEST_COMPLETION_PARAMS` или вызывает метод `wdfRequestGetCompletionParams`, чтобы получить указатель на эту структуру.

Эта структура содержит все параметры запроса.

Параметры завершения остаются действительными до тех пор, пока запрос не будет завершен текущим драйвером, удален, повторно использован или переформатирован. Выполнение любой из только что перечисленных операций делает указатель недействительным.

### Подсказка

Возможно, вы задаете себе вопрос, в чем состоит разница между параметрами завершения — отправленными в виде параметра функции обратного вызова завершения ввода/вывода — и параметрами запроса. Параметры запроса — это те параметры, которые поступают драйверу вместе с запросом, а параметры завершения запроса — это параметры, которые драйвер посыпает вместе с запросом получателю ввода/вывода. В сущности, если запрос не модифицируется никаким образом, то параметры завершения отсутствуют. В этом случае, параметры запроса служат также в качестве параметров завершения запроса.

## Отправление запросов ввода/вывода

Драйвер может отправлять запросы ввода/вывода или синхронно, или асинхронно, а также может указать значение тайм-аута для запроса. По истечении периода тайм-аута инфраструктура отменяет запрос. Драйверы отправляют запросы с помощью следующих методов:

- ◆ драйверы UMDF вызывают метод `IWDFIoRequest::Send` объекта запроса;
- ◆ драйверы KMDF вызывают метод `wdfRequestSend`, передавая ему дескриптор объекта запроса.

Чтобы отправить запрос синхронно, вместо метода `WdfRequestSend` драйвер KMDF может вызвать один из методов семейства `WdfIoTargetSendXxxSynchronously`.

Драйвер указывает на получатель запроса ввода/вывода. В случае нестандартного получателя ввода/вывода драйвер должен предварительно создать и открыть его. Инфраструктура получает ссылку на объект запроса, с тем, чтобы предотвратить удаление ассоциированных ресурсов, пока запрос ожидает отправки объекту устройства получателя.

KMDF также поддерживает методы для форматирования запроса и синхронной его отправки в одной операции.

Драйвер KMDF может отправить синхронный запрос ввода/вывода, используя один из методов семейства `WdfIoTargetSendXxxSynchronously`, которые форматируют и отправляют запрос за один вызов метода. Для форматирования и отправки синхронных запросов ввода/вывода применяются следующие методы KMDF:

- ◆ `WdfIoTargetSendInternalIoctlSynchronously` — форматирует внутренний запрос IOCTL для любого получателя ввода/вывода, отправляет его получателю и возвращает управление по завершению запроса получателем;
- ◆ `WdfIoTargetSendInternalIoctlOthersSynchronously` — форматирует нестандартный внутренний запрос IOCTL для любого получателя ввода/вывода, отправляет его получателю и возвращает управление по завершению запроса получателем;
- ◆ `WdfIoTargetSendIoctlSynchronously` — форматирует запрос IOCTL для любого получателя ввода/вывода, отправляет его получателю и возвращает управление по завершению запроса получателем;
- ◆ `WdfIoTargetSendReadSynchronously` — форматирует запрос на чтение для любого получателя ввода/вывода, отправляет его получателю и возвращает управление по завершению запроса получателем;
- ◆ `WdfIoTargetSendWriteSynchronously` — форматирует запрос на запись для любого получателя ввода/вывода, отправляет его получателю и возвращает управление по завершению запроса получателем.

Этим методам передаются те же самые параметры, как и для соответствующих методов семейства `WdfIoTargetFormatRequestXxx`, а также следующие два дополнительных параметра:

- ◆ параметр флагов, как описывается в следующем разделе;
- ◆ выходной параметр, в котором инфраструктура возвращает количество переданных байтов.

## Опция для отправки запросов

Когда драйвер отправляет запрос ввода/вывода, он указывает получателя этого запроса. Драйвер также может указать значение тайм-аута и любые флаги, которые управляют, каким образом инфраструктура отправляет запрос.

Далее приводится сводка возможных флагов для отправки запросов ввода/вывода:

- ◆ `WDF_REQUEST_SEND_OPTION_TIMEOUT` — отменяет запрос, когда истекает тайм-аут;
- ◆ `WDF_REQUEST_SEND_OPTION_SYNCHRONOUS` — отсылает запрос синхронно;
- ◆ `WDF_REQUEST_SEND_OPTION_IGNORE_TARGET_STATE` — отправляет запрос, несмотря на то, позволяет это или нет состояние получателя ввода/вывода;
- ◆ `WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET` — отправляет запрос асинхронно, без функции обратного вызова для завершения запроса ввода/вывода.

В следующем разделе приводится дополнительная информация о значении каждого из флагов.

## Значения тайм-аута для запросов ввода/вывода

При отправке запроса драйвер может обозначить значение тайм-аута, указывающее, сколько времени инфраструктура должна ожидать завершения запроса, по истечению которого она отменяет его. Драйвер выражает значение тайм-аута отрицательным числом с интервалом в 100 наносекунд. Таким образом, для указания тайм-аута в 10 секунд драйвер устанавливает значение тайм-аута в  $-100\ 000\ 000$ . Если драйвер предоставляет значение тайм-аута, он также должен установить флаг `WDF_REQUEST_SET_OPTION_TIMEOUT`.

Драйверы KMDF могут использовать предоставляемые инфраструктурой функции преобразования времени, что позволяет получить намного больше удобочитаемый код.

Соотношение периода отмены запроса и значения тайм-аута приведено в табл. 9.8.

**Таблица 9.8. Значение тайм-аута в зависимости от периода отмены запроса**

Период отмены запроса	Значение тайм-аута
$N$ секунд после того, как инфраструктура получит запрос	Отрицательное число, равное $-N$ , умноженное на 10 000 000. Или <code>WDF_REL_TIMEOUT_IN_SEC(N)</code> . (Только в KMDF.)
Никогда	Ноль и неустановленный флаг <code>WDF_REQUEST_SET_OPTION_TIMEOUT</code>
Немедленно	Ноль и установленный флаг <code>WDF_REQUEST_SET_OPTION_TIMEOUT</code>

Применение драйверами KMDF функций преобразования времени, предоставляемых инфраструктурой, описывается в главе 12.

## Синхронные и асинхронные запросы ввода/вывода

По умолчанию WDF отправляет запросы ввода/вывода асинхронно. Управление возвращается драйверу сразу же после того, как инфраструктура ставит запрос в очередь на отправку получателю ввода/вывода. Чтобы получить извещение о завершении запроса, драйвер должен зарегистрировать функцию обратного вызова для завершения запроса. Драйвер должен отправлять запросы ввода/вывода асинхронно всегда, когда это возможно, с тем, чтобы избежать блокирования потока, в котором он исполняется.

При отправке запросов асинхронно, метод отправки возвращает управление немедленно. Чтобы определить, смогла ли инфраструктура поставить запрос в очередь на доставку получателю ввода/вывода, драйвер должен проверить следующее значение:

- ◆ в драйверах UMDF метод `Send` возвращает значение `HRESULT`. При успешной операции отправки возвращается значение `S_OK`. При неудачной операции возвращается код ошибки, указывающий причину неудачи;
- ◆ в драйверах KMDF метод `WdfRequestSend` возвращает булево значение. Если возвращается значение `FALSE`, то драйвер должен вызвать метод `WdfRequestGetStatus`, чтобы определить причину неудачи.

Но в некоторых ситуациях драйвер должен отправлять запросы синхронно. Например, если аппаратное обеспечение устройства требует вмешательства, прежде чем драйвер может обрабатывать входящий запрос ввода/вывода, он должен послать запрос IOCTL, чтобы изменить состояние устройства. Для отправки запроса синхронно драйвер устанавливает флаг `WDF_REQUEST_SEND_OPTION_SYNCHRONOUS`.

При отправке запроса синхронно поток драйвера блокируется до тех пор, пока получатель ввода/вывода не завершит запрос. Для того чтобы избежать неприемлемых задержек при обработке, драйверам следует устанавливать тайм-аут для синхронных запросов ввода/вывода, если это возможно.

## Эффект, оказываемый состоянием получателя ввода/вывода

По умолчанию инфраструктура WDF отправляет запросы ввода/вывода получателю ввода/вывода только тогда, когда получатель ввода/вывода пребывает в состоянии `Started`. Драйвер может переопределить это поведение, указав флаг `WDF_REQUEST_SEND_OPTION_IGNORE_TARGET_STATE`. В таком случае инфраструктура ставит запрос в очередь на доставку получателю ввода/вывода, даже если последний находится в состоянии `Stopped`.

В табл. 9.9 приведено краткое описание, что происходит при отправке драйвером запросов получателю ввода/вывода при разных состояниях получателя.

**Таблица 9.9. Эффект состояния получателя ввода/вывода на запрос ввода/вывода**

Состояние получателя	Действие инфраструктуры
Started (Запущен)	Отсылает запрос ввода/вывода
Stopped (Остановлен)	По умолчанию ставит запрос ввода/вывода в очередь. Если драйвер установил флаг <code>WDF_REQUEST_SEND_OPTION_IGNORE_TARGET_STATE</code> , отсылает запрос без постановки его в очередь
Stopped for query-remove (Остановлен для удаления по результатам опроса)	Не выполняет операцию отправления

**Таблица 9.9 (окончание)**

Состояние получателя	Действие инфраструктуры
Closed (Закрыт)	Не выполняет операцию отправления
Deleted (Удален)	Не выполняет операцию отправления. Если объект получателя ввода/вывода уже был удален и дескриптор получателя больше не действителен, драйвер UMDF может сгенерировать останов драйвера, а драйвер KMDF — останов bugcheck

Если устройство получателя было остановлено, но не удалено, инфраструктура ставит запрос в очередь, для отправки его позже, когда устройство получателя возобновит работу. Если драйвер WDF указывает значение тайм-аута, таймер запускается при добавлении запроса к очереди.

Если устройство получателя было удалено, попытка отправить запрос завершается неудачей. Драйвер может определить причину неудачного завершения отправки, вызывая инфраструктуру, чтобы получить статус завершения запроса. Возвращаемое значение STATUS\_INVALID\_DEVICE\_STATE или HRESULT\_FROM\_NT (STATUS\_INVALID\_DEVICE\_STATE) указывает, что получатель ввода/вывода не находился в состоянии Started.

### Опция "отправил и забыл"

Когда драйвер отправляет запрос асинхронно, он обычно регистрирует функцию обратного вызова завершения ввода/вывода. Но в некоторых случаях драйвер не использует результаты запроса ввода/вывода и не требует статуса завершения запроса. Например, после отправки запроса вниз по стеку драйверу, фильтрующему ввод, может больше не требоваться выполнять дополнительную обработку этого запроса.

Такой драйвер при отправке запроса должен установить флаг WDF\_REQUEST\_SEND\_OPTION\_SEND\_AND\_FORGET и не регистрировать функцию обратного вызова завершения ввода/вывода. Драйверы должны устанавливать этот флаг только для запросов, получаемых ими от инфраструктуры, но не для запросов, создаваемых самими драйверами. Этот флаг эквивалентен функции IoSkipCurrentIrpStackLocation, которую драйверы WDM вызывают для отправки не модифицированного запроса ввода/вывода вниз по стеку.

Если драйвер устанавливает флаг "отправил и забыл", инфраструктура не ведет учет информации об объекте запроса в своем списке отправленных или поставленных в очередь запросов. Кроме этого, изменения в состоянии объекта получателя ввода/вывода не влияют на запрос. Поэтому, если получатель ввода/вывода остановлен, закрыт или удален, объект получателя ввода/вывода не отменяет запрос, с тем, чтобы он мог оставаться активным для драйвера получателя.

Для драйверов UMDF, в случае неожиданного прекращения исполнения хост-процесса, отражатель может всегда найти и отменить запрос.

## Пример UMDF: отправка запроса стандартному получателю ввода/вывода

Образец драйвера USB Filter демонстрирует, как драйвер UMDF может отфильтровать запрос ввода/вывода, после чего отправить его вниз по стеку стандартному получателю ввода/вывода. Драйвер инвертирует биты в запросах на чтение и запись.

Чтобы получить указатель на интерфейс для объекта стандартного получателя ввода/вывода, драйвер вызывает метод `GetDefaultIoTarget` инфраструктурного объекта устройства следующим образом:

```
FxDevice->GetDefaultIoTarget(&m_FxIoTarget);
```

Инфраструктура создает стандартный получатель ввода/вывода при создании ею объекта устройства, поэтому драйвер может вызывать метод `GetDefaultIoTarget` в любое время, когда у него есть достоверный указатель на интерфейс инфраструктурного объекта устройства. Драйвер USB Filter вызывает этот метод после того, как он успешно создаст стандартную очередь ввода/вывода.

В листинге 9.9 показано, каким образом драйвер USB Filter отправляет запрос стандартному получателю ввода/вывода. Фрагмент кода, показанный в этом листинге, был взят из файла Queue.cpp.

#### Листинг 9.9. Отправка запроса стандартному получателю ввода/вывода в драйвере UMDF

```
void CMyQueue::ForwardRequest(
    in IWDFIoRequest* FxRequest )
{
    // Регистрируется обратный вызов для завершения.
    IRequestCallbackRequestCompletion *completionCallback =
        QueryIRequestCallbackRequestCompletion();
    FxRequest->SetCompletionCallback(completionCallback, NULL);
    completionCallback->Release();
    // Запрос форматируется для пересылки.
    FxRequest->FormatUsingCurrentType( );
    // Отправка запроса стандартному исполнителю ввода/вывода
    HRESULT hrSend = S_OK;
    hrSend = FxRequest->Send( m_FxIoTarget, // Стандартный исполнитель
                                // ввода/вывода
                                0,           // Флага нет.
                                0           // Тайм-аута нет.
                           );
    // Если отправка завершилась неудачей, запрос завершается
    // со статусом неудачи.
    if (FAILED(hrSend)) {
        FxRequest->CompleteWithInformation(hrSend, 0);
    }
    return;
}
```

Приведенная в листинге 9.9 функция `ForwardRequest` выполняет следующие три задачи:

- ◆ регистрирует обратный вызов для завершения;
- ◆ форматирует запрос для получателя ввода/вывода;
- ◆ отправляет запрос получателю ввода/вывода.

Чтобы зарегистрировать обратный вызов для завершения запроса, драйвер запрашивает указатель на свой интерфейс `IRequestCallbackRequestCompletion` и передает этот указатель методу `IWDFIoRequest::SetCompletionCallback`, после чего освобождает ссылку на интерфейс.

Дальше драйвер вызывает метод `IWDFIoRequest::FormatUsingCurrentType`, чтобы подготовить запрос для отправки получателю ввода/вывода. Этот метод организует следующую ячейку

стека ввода/вывода для драйвера получателя в нижележащем пакете IRP. Если драйвер не установит флаг `WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET`, он должен вызвать один из методов форматирования, чтобы организовать следующую ячейку стека ввода/вывода.

Наконец, драйвер отправляет запрос, вызывая метод `IWDFIoRequest::Send`. Метод `Send` принимает следующие три параметра: указатель на интерфейс для стандартного получателя ввода/вывода, набор флагов и значение тайм-аута. Для параметров флагов и тайм-аута драйвер передает нулевые значения. Нулевое значение флага означает следующее:

- ◆ драйвер не указывает тайм-аут для запроса;
- ◆ инфраструктура отправляет пакет асинхронно;
- ◆ инфраструктура отправляет пакет, только если получатель находится в действующем состоянии, т. е., не остановлен и не удален;
- ◆ драйвер зарегистрировал для запроса процедуру обратного вызова завершения ввода/вывода.

После завершения запроса нижерасположенными драйверами в стеке, инфраструктура вызывает метод `OnCompletion` интерфейса `IRquestCallbackRequestCompletion` драйвера. Исходный код этого метода показан в листинге 9.10.

#### Листинг 9.10. Завершение отправленного запроса в драйвере UMDF

```
Void CMYQueue::OnCompletion(
    /* [in] */ IWDFIoRequest* FxRequest,
    /* [in] */ WDFIoTarget* FxIoTarget,
    /* [in] */ IWDFRequestCompletionParams* CompletionParams,
    /* [in] */ PVOID Context )
{
    // Для запроса на чтение инвертируем прочитанные биты.
    if (WdfRequestRead == FxRequest->GetType()) {
        . . . // Код опущен, чтобы сохранить место.}
    else {
        // Завершаем объект запроса с теми же параметрами,
        // с которыми он был завершен нижерасположенными драйверами.
        FxRequest->CompleteWithInformation(
            CompletionParams->GetCompletionStatus(),
            CompletionParams->GetInformation());
    }
}
```

Для запроса на чтение функция обратного вызова завершения ввода/вывода обрабатывает возвращенные данные, инвертируя биты, после чего завершает запрос. Для запросов другого типа функция обратного вызова просто вызывает метод `IWDFIoRequest::CompleteWithInformation`. Функция получает статус завершения запроса и количество переданных байтов, вызывая для этого методы на интерфейсе `IWDFRequestCompletionParams`, и передает эти значения методу `CompleteWithInformation`.

## Пример KMDF: опция "отправил и забыл"

Образец KMDF-драйвера Kbfiltr фильтрует данные, вводимые с клавиатуры, после чего отправляет получаемые внутренние запросы IOCTL вниз по стеку стандартному получателю ввода/вывода. Фрагмент кода, показанный в этом листинге, был взят из файла Kbfiltr.c.

Чтобы получить указатель на стандартный получатель ввода/вывода, драйвер вызывает метод `WdfDeviceGetIoTarget` следующим образом:

```
KbFilter_ForwardRequest(Request, WdfDeviceGetIoTarget(hDevice));
```

Инфраструктура создает и инициализирует стандартный получатель ввода/вывода при создании его объекта устройства, поэтому драйвер может вызывать метод `WdfDeviceGetIoTarget` в любое время, когда у него есть действительный указатель на инфраструктурный объект устройства. Образец драйвера `Kbfilt` вызывает этот метод в своей процедуре обратного вызова `EvtIoInternalDeviceControl` после анализа прибывшего запроса.

В листинге 9.11 показано, каким образом драйвер `Kbfilt` отправляет запрос стандартному получателю ввода/вывода.

#### Листинг 9.11. Использование опции "отправил и забыл" в драйвере KMDF

```
VOID KbFilter_ForwardRequest(IN WDFREQUEST Request,
                             IN WDFIOTARGET Target)
{
    WDF_REQUEST_SEND_OPTIONS options;
    BOOLEAN ret;
    NTSTATUS status;
    WDF_REQUEST_SEND_OPTIONS_INIT(&options,
                                   WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET);
    ret = WdfRequestSend(Request, Target, &options);
    if (ret == FALSE) {
        status = WdfRequestGetStatus(Request);
        WdfRequestComplete(Request, status);
    }
    return;
}
```

В этой функции метод инициализирует структуру `WDF_REQUEST_SEND_OPTIONS`, после чего отправляет запрос. Установив опцию `WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET`, драйвер дает указание инфраструктуре отправить запрос асинхронно, не предоставляя извещения, был ли запрос завершен или отменен. Драйверы не должны завершать запросы, отправляемые ими с этой опцией; также они не могут регистрировать функции обратного вызова завершения ввода/вывода для таких запросов.

После инициализации опций драйвер отправляет запрос стандартному получателю ввода/вывода, вызывая для этого метод `WdfRequestSend`. В качестве параметров методу передается дескриптор объекта, дескриптор получателя ввода/вывода и указатель на структуру `WDF_REQUEST_SEND_OPTIONS`.

Если инфраструктура не отправит запрос, метод `WdfRequestSend` возвращает `FALSE`, и драйвер вызывает метод `WdfRequestGetStatus`, чтобы выяснить причину неудачи отправки, и метод `WdfRequestComplete`, чтобы завершить запрос со статусом неудачи.

В случае успешной отправки запроса инфраструктурой метод `WdfRequestSend` возвращает `TRUE`, и на этом обработка драйвером запроса ввода/вывода завершается.

#### Примечание

В данном примере драйвер не вызывает метод для форматирования перед отправкой запроса, т. к. он устанавливает флаг `WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET`.

## Пример KMDF: форматирование и отправление запроса получателю ввода/вывода

В листинге 9.12 показан пример, как драйвер KMDF форматирует и отправляет запрос ввода/вывода получателю ввода/вывода. Код примера взят из файла Toastmon.c. Образец драйвера Toastmon создает и открывает удаленный получатель ввода/вывода и предварительно выделяет объекты запроса ввода/вывода для запросов на чтение и запись. В качестве родителя драйвер устанавливает объект получателя ввода/вывода для обоих объектов запроса.

**Листинг 9.12. Форматирование и отправка запроса на чтение драйвером KMDF**

```
NTSTATUS ToastMon_PostReadRequests(IN WDFIOTARCET IoTarget)
{
    WDFREQUEST request;
    NTSTATUS status;
    PTARCKET_DEVICE_INFO targetInfo;
    WDFMEMORY memory;
    WDF_OBJECT_ATTRIBUTES attributes;
    targetInfo = GetTargetDeviceInfo(IoTarget);
    request = targetInfo->ReadRequest;
    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    status = WdfMemoryCreate(&attributes,
                           NonPagedPool,
                           DRIVER_TAG,
                           READ_BUF_SIZE,
                           &memory,
                           NULL); // buffer pointer
    if (!NT_SUCCESS(status)) return status;
    status = WdfIoTargetFormatRequestForRead(IoTarget,
                                             request,
                                             memory,
                                             NULL, // Buffer offset
                                             NULL); // OutputBufferOffset
    if (!NT_SUCCESS(status)) return status;
    WdfRequestSetCompletionRoutine(request,
                                   Toastmon_ReadRequestCompletionRoutine,
                                   targetInfo);
    // Считываем поле ReadRequest в области контекста.
    targetInfo->ReadRequest = NULL;
    if (WdfRequestSend(request, IoTarget, WDF_NO_SEND_OPTIONS) == FALSE) {
        status = WdfRequestGetStatus(request);
        targetInfo->ReadRequest = request;
    }
    return status;
}
```

Функция ToastMon\_PostReadRequests в листинге 9.12 форматирует запросы на чтение и отправляет их удаленному получателю ввода/вывода. В качестве параметра функции передается дескриптор получателя ввода/вывода. Драйвер ранее сохранил дескриптор объекта запроса ввода/вывода в области контекста объекта получателя ввода/вывода, поэтому функция первым делом получает указатель на область контекста, вызывая функцию доступа (т. е.

GetTargetDeviceInfo), после чего извлекает объект запроса ввода/вывода из области контекста.

Дальше драйвер инициализирует структуру атрибутов объекта. Он передает эту структуру методу `WdfMemoryCreate` для создания нового объекта памяти WDF и выделения для запроса нового буфера из нестраничной памяти.

Если операции создания объекта памяти и буфера выполняются успешно, драйвер форматирует запрос на чтение, вызывая для этого метод `WdfIoTargetFormatRequestForRead`, который принимает четыре параметра. В качестве первых двух параметров драйвер передает этому методу дескриптор объекта запроса ввода/вывода и дескриптор объекта памяти. Для последних двух параметров, указывающих смещения в буфере вывода и в устройстве, передаются значения `NULL`.

Отформатировав запрос ввода/вывода, драйвер регистрирует функцию обратного вызова завершения ввода/вывода, с тем, чтобы он мог получить результаты выполнения запроса. Эта функция вызывается инфраструктурой, когда получатель ввода/вывода завершает запрос. После этого драйвер удаляет дескриптор запроса из области контекста и отправляет запрос получателю ввода/вывода, вызывая для этого метод `WdfRequestSend`.

При вызове метода `WdfRequestSend` драйвер передает ему дескриптор объекта запроса, дескриптор объекта получателя ввода/вывода и значение `WDF_NO_SEND_OPTIONS`, указывающее, что драйвер для отправки запроса ввода/вывода принимает все настройки по умолчанию инфраструктуры WDF. То есть, инфраструктура отправляет запрос асинхронно, без значения тайм-аута и завершает запрос неудачей, если получатель ввода/вывода остановлен. Метод `WdfRequestSend` возвращает булево значение, указывающее статус постановки запроса в очередь к получателю ввода/вывода. Если отправление закончилось неудачей или если инфраструктура не поставила запрос в очередь, драйвер вызывает метод `WdfRequestGetStatus`, чтобы получить код ошибки, и устанавливает объект отформатированного запроса в области контекста. После этого функция `ToastMon_PostReadRequests` возвращает управление.

Хотя данный образец драйвера использует удаленный получатель ввода/вывода, отправление драйверами запросов стандартному получателю ввода/вывода производится почти таким же образом. Единственная разница заключается в том, что вместо создания и открытия удаленного получателя ввода/вывода драйвер вызывает инфраструктуру, чтобы получить дескриптор стандартного получателя ввода/вывода.

## Разбивка запросов ввода/вывода на подзапросы

Если полученный драйвером запрос требует считать или записать больший объем данных, чем позволяют технические возможности аппаратного обеспечения, драйвер может разбить первоначальный запрос на несколько меньших подзапросов.

Например, допустим, что получатель ввода/вывода за одну операцию может обработать самое большое `MAX_BYTES` байтов данных. Если драйвер получит запрос ввода/вывода на чтение больше чем `MAX_BYTES` байтов, он может разбить этот запрос, выполнив следующие действия:

1. Извлечь объект памяти из объекта входящего запроса ввода/вывода.
2. Создать новый объект запроса ввода/вывода или использовать повторно уже существующий объект ввода/вывода, созданный ранее драйвером. (Повторно использовать объекты ввода/вывода могут только драйверы KMDF.)

3. Определить количество байтов и смещение в буфере для данного подзапроса.

В первом подзапросе обычно указывается размер буфера в `MAX_BYTES` и нулевое смещение. Во втором подзапросе указывается смещение в `MAX_BYTES` и размер буфера в `MAX_BYTES` или равный количеству байтов, оставшихся считать или записать, что меньше. Любые дополнительные подзапросы создаются, следуя этому шаблону.

4. Отформатировать объект подзапроса существующим объектом памяти из полученного запроса и подсчитанной длиной буфера и смещением.
5. Отправить запрос ввода/вывода.
6. Создать, отформатировать и отправить дополнительные подзапросы как требуется для завершения ввода/вывода.
7. Завершить первоначальный полученный запрос после завершения всех составляющих его подзапросов.

Объект памяти — и, таким образом, буфер, содержащийся в нем, — уже содержит все требуемые выходные данные.

Чтобы извлечь объект памяти из объекта входящего запроса ввода/вывода:

- ◆ драйверы UMDF вызывают метод `IWDFRequest::GetInputMemory` или `GetOutputMemory`;
- ◆ драйверы KMDF вызывают метод `WdfRequestRetrieveInputMemory` или `WdfRequestRetrieveOutputMemory`.

Если драйвер использует объект памяти из входящего запроса ввода/вывода в каждом подзапросе, то буфер уже содержит все выходные данные из получателя ввода/вывода. Драйверу не требуется копировать данные в буфер перед тем, как завершить первоначальный, входящий запрос ввода/вывода. Но при вызове инфраструктуры для завершения запроса он должен заполнить статус завершения ввода/вывода и количество переданных байтов.

Но если драйвер еще не завершил первоначальный запрос, инфраструктура удерживает для него ссылку на объект памяти в получателе ввода/вывода. Прежде чем завершить первоначальный запрос, драйвер должен освободить эту ссылку следующим образом:

- ◆ драйвер UMDF должен реализовать интерфейс `IRquestCallbackRequestCompletion` на подзапросе и в методе `OnCompletion` вызывать метод `Release` на объекте памяти;
- ◆ драйвер KMDF должен зарегистрировать функцию обратного вызова `CompletionRoutine` для подзапроса, и эта функция должна вызвать метод `WdfRequestReuse` на подзапросе.

Если перед завершением первоначального запроса драйвер не вызовет метод `WdfRequestReuse`, то инфраструктура сгенерирует останов `bugcheck` по причине оставшихся ссылок на объект памяти в первоначальном запросе.

## Пример KMDF: повторное использование объекта ввода/вывода

Если драйвер KMDF отправляет большое количество запросов ввода/вывода, его можно создать таким образом, чтобы он предварительно выделял один или несколько объектов запроса ввода/вывода и использовал эти объекты вместо создания нового объекта запроса ввода/вывода для каждого нового запроса. Повторное использование объектов ввода/вывода может способствовать улучшению производительности, особенно если драйвер отправляет большое количество асинхронных запросов. Так как в таком случае память уже выделена,

драйверу не требуется тратить время на выделение памяти для каждого нового запроса. Отправляя запросы асинхронно, драйвер не блокирует поток, пока он ожидает ответа.

Драйвер KMDF может повторно использовать объект запроса ввода/вывода, только если объект был создан методом `WdfRequestCreate` или `WdfRequestCreateFromIrp`. Объект запроса ввода/вывода, полученный драйвером от инфраструктуры, не может быть повторно использован.

Прежде чем повторно использовать объект запроса ввода/вывода, драйвер должен извлечь из него результаты предыдущего запроса и полностью окончить обработку этого запроса. Потом, после того как процесс завершения запроса вернулся обратно к драйверу, для того чтобы повторно использовать объект запроса ввода/вывода, драйвер вызывает метод `WdfRequestReuse`. Метод `WdfRequestReuse` повторно инициализирует объект запроса, что делает всю ранее содержащуюся в нем информацию недоступной для драйвера. Поэтому драйвер должен вызывать этот метод только в том случае, если объект запроса ввода/вывода не содержит требуемую драйверу информацию. Прежде чем отправить запрос ввода/вывода опять, драйвер должен отформатировать его и зарегистрировать функцию обратного вызова для завершения запроса.

### Внимание!

Драйвер может повторно инициализировать объект запроса в своей функции обратного вызова завершения запроса, но в некоторых случаях не должен отправлять следующий запрос в том же самом вызове этой функции. Такое действие может вызвать рекурсию и, если окончится системный стек, последующий системный сбой.

Если нижерасположенные драйверы завершат запрос синхронно с отправкой запроса, инфраструктура может вызвать функцию обратного вызова завершения ввода/вывода снова до того, как предыдущий вызов завершит обработку первого запроса.

В листинге 9.13 приведен пример повторного использования объекта запроса ввода/вывода драйвером `Ndisedge` для отправки асинхронного запроса IOCTL. Этот код взят из файла `Ndisedge\60\Request.c`.

#### Листинг 9.13. Повторное использование объекта запроса ввода/вывода в драйвере KMDF

```
NTSTATUS NICSendOidRequestToTargetAsync(
    IN WDFIOTARGET          IoTarget,
    IN WDFREQUEST           Request,
    IN PFILE_OBJECT         FileObject,
    IN ULONGNC              IoctlControl Code,
    IN OUT PVOID             InputBuffer,
    IN ULONGNC              InputBufferLength,
    IN OUT PVOID             OutputBuffer,
    IN ULONGNC              OutputBufferLength,
    OUT PULONGNC            BytesReadOrWritten)
{
    NTSTATUS                  status;
    PREQUEST_CONTEXT          reqContext;
    WDF_REQUEST_REUSE_PARAMS  params;
    WDFMEMORY                 inputMem, outputMem;
    UNREFERENCED_PARAMETER(FileObject);

    WDF_REQUEST_REUSE_PARAMS_INIT(&params,
        WDF_REQUEST_REUSE_NO_FLACS,
        STATUS_SUCCESS);
```

```

status = WdfRequestReuse(Request, &params);
if (!NT_SUCCESS(status)) return status;
// Назначаем новые буферы предварительно выделенной объектам памяти.
reqContext = GetRequestContext(Request);
inputMem = outputMem = NULL;
if (InputBuffer != NULL) {
    status = WdfMemoryAssignBuffer(reqContext->InputMemory,
                                   InputBuffer,
                                   InputBufferLength);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    inputMem = reqContext->InputMemory;
}
if (OutputBuffer != NULL) {
    status = WdfMemoryAssignBuffer(reqContext->OutputMemory,
                                   OutputBuffer,
                                   OutputBufferLength);
    if (!NT_SUCCESS(status)) return status;
    outputMem = reqContext->OutputMemory;
}
status = WdfIoTargetFormatRequestForIoctl(IoTarget,
                                         Request,
                                         IoctlControlCode,
                                         inputMem,
                                         NULL, //InputBufferoffsets
                                         outputMem,
                                         NULL //OutputBufferOffset );
if (!NT_SUCCESS(status)) return status;
WdfRequestSetCompletionRoutine(Request,
                               NICSendOidRequestToTargetAsyncCompletionRoutine,
                               BytesReadOrWritten);
if (WdfRequestSend (Request, IoTarget, WDF_NO_SEND_OPTIONS) == FALSE) {
    status = WdfRequestGetStatus(Request);
}
return status;
}

```

Драйвер не должен предпринимать попытку повторно использовать объект запроса ввода/вывода до тех пор, пока предварительный запрос не будет завершен. По этой причине, многие драйверы вызывают метод `WdfRequestReuse` для повторной инициализации объекта запроса ввода/вывода из своих функций обратного вызова завершения ввода/вывода. Но драйвер, показанный в листинге 9.13, работает по-иному.

Причиной этому является то обстоятельство, что этот драйвер отправляет запросы ввода/вывода из рабочего элемента, который ожидает на событие, установленное обратным вызовом завершения ввода/вывода.

Поэтому, хотя драйвер отправляет запросы ввода/вывода асинхронно, его конструкция обеспечивает, что когда исполняется метод `NICSendOidRequestToTargetAsync`, предыдущий запрос ввода/вывода будет уже завершен. Чтобы избежать предоставления специального кода для обработки первого запроса, этот драйвер вызывает метод `WdfRequestReuse` также и на новом созданном запросе. В это время запрос не содержит никакой полезной информации, поэтому вызов метода повторного использования не приносит никакого вреда.

Прежде чем драйвер вызовет метод `WdfRequestReuse`, он должен инициализировать структуру `WDF_REQUEST_REUSE_PARAMS` для передачи этому методу в качестве параметра. Эта структура содержит набор флагов и начальное значение `NTSTATUS` для установки в запросе. В настоящее время единственный возможный флаг применим только к объектам запроса ввода/вывода, созданным вызовом драйвером метода `WdfRequestCreateFromIrp`. Так как драйвер `Ndisedge` не создал запрос из пакета IRP, то он не указывает никаких флагов и указывает `STATUS_SUCCESS`.

Дальше драйвер вызывает метод `WdfRequestReuse`, передавая ему в качестве параметров дескриптор объекта запроса ввода/вывода и указатель на структуру параметров для повторного использования.

При успешном завершении метода `WdfRequestReuse` драйвер организовывает буферы для запроса ввода/вывода. Ранее драйвер вызвал метод `WdfMemoryCreatePreallocated`, чтобы создать объекты памяти WDF для буферов ввода и вывода, и сохранил дескрипторы этих объектов в области контекста объекта запроса. Доступ к области контекста можно получить с помощью указателя, возвращаемого функцией доступа драйвера, называемой `GetRequestContext`.

Если клиент, вызывающий процедуру `NICSendOidRequestToTargetAsync`, предоставил действительный указатель в `InputBuffer`, то драйвер вызывает метод `WdfMemoryAssignBuffer`, чтобы ассоциировать этот буфер с объектом `InputMemory`, который был сохранен в области контекста запроса. Драйвер повторяет этот шаг для `OutputBuffer`.

Организовав входные и выходные буферы, драйвер форматирует запрос ввода/вывода, вызывая для этого метод `WdfIoTargetFormatRequestForIoctl`. В качестве параметров драйвер передает методу дескрипторы получателя ввода/вывода и объекта запроса ввода/вывода, а также код IOCTL, переданный вызывающим клиентом, и дескрипторы входного и выходного объектов памяти. В случае успешного выполнения этого метода, запрос почти готовый к отправке.

Задачей, оставшейся выполнить драйверу перед отправкой запроса ввода/вывода, является регистрация обратного вызова завершения запроса. Драйвер выполняет эту задачу, вызывая метод `WdfRequestSetCompletionRoutine`, после чего вызывает метод `WdfRequestSend` для отправки запроса.

При вызове метода `WdfRequestSend` драйвер передает ему значение `WDF_NO_SEND_OPTIONS`, указывающее, что для отправки запроса ввода/вывода драйвер принимает все настройки по умолчанию.

Если получатель ввода/вывода не находится в состоянии `Started`, метод `WdfRequestSend` возвращает `FALSE`, и драйвер вызывает метод `WdfRequestGetStatus`, чтобы определить причину неудачи. В противном случае функция `NICSendOidRequestToTargetAsyn` возвращает управление. Когда получатель ввода/вывода завершает запрос, инфраструктура вызывает установленную драйвером функцию обратного вызова завершения запроса, в которой драйвер извлекает результаты запроса.

## Отмена отправленного запроса

Если устройство получателя останавливается или если оно было неожиданно удалено, драйверу может быть необходимо отменить запросы ввода/вывода, которые он уже отправил получателю ввода/вывода. Для этого:

- ◆ чтобы отменить один запрос, драйвер UMDF вызывает метод `IWDFIoRequest::CancelSentRequest`.

А чтобы отменить все запросы, отправленные в определенный файл, драйвер UMDF вызывает метод `IWDFIoTarget::CancelSentRequestsForFile`;

- ◆ драйвер KMDF вызывает метод `WdfRequestCancelSentRequest`.

Если запрос был поставлен в очередь для доставки получателю ввода/вывода, но еще не был доставлен, инфраструктура отменяет запрос. Если запрос был переслан другому драйверу, его можно отменить, только если этот драйвер поддерживает отмену запросов.

В любое время при отмене запроса ввода/вывода может возникнуть состояние гонок. Запрос, может быть, уже был завершен — и, поэтому, объект запроса WDF удален — в период времени между вызовом драйвером метода отмены и началом исполнения этого метода. Но до тех пор, пока существует ссылка на объект запроса WDF, его дескриптор остается действительным, и поэтому в большинстве случаев драйверу не требуется организовывать синхронизацию для вызова метода отмены. Прежде чем отменить нижележащий в стеке пакет IRP, инфраструктура удостоверяется в том, что он все еще действителен.

### **Отмена запросов инфраструктурой UMDF**

В Windows Vista и более поздних версиях для отмены отдельного запроса, когда драйвер вызывает метод `CancelSentRequest`, UMDF использует функцию Windows `CancelIoEx`.

Но более ранние версии Windows не поддерживают метод `CancelIoEx`. В этих версиях UMDF реализует отмену запросов с помощью отражателя для тех запросов, которые стек устройств UMDF пересыпает объекту устройства Down через стандартный получатель ввода/вывода. Но в этих версиях Windows UMDF не может отменить индивидуально новые запросы, созданные драйвером UMDF, хотя драйвер может отменить все отправленные им запросы.

### **Пример UMDF: отмена всех запросов ввода/вывода для файла**

Драйвер UMDF обычно отменяет все ожидающие исполнения запросы ввода/вывода в своем обратном вызове метода `IFileCallbackCleanup::OnCleanupFile`. Инфраструктура вызывает этот метод, когда приложение вызывает функцию Windows `CloseHandle`.

В листинге 9.14 показан пример отмены драйвером `Usb\Echo_driver` всех запросов, которые он отправил определенному файловому получателю. Этот драйвер реализует интерфейс `IFileCallbackCleanup` на объекте устройства. Метод `OnCleanupFile` объекта устройства просто вызывает показанный в листинге 9.14 метод `OnCleanupFile`, который реализован на объекте обратного вызова стандартной очереди. Соответственно, файл, из которого взят код примера, называется `Queue.cpp`.

#### **Листинг 9.14. Отмена всех запросов файлового ввода/вывода в драйвере UMDF**

```
Void CMyQueue::OnCleanupFile(
    /*[in]*/ IWDFFile* pFileObject
)
{
    m_Parent->GetInputPipe()->CancelSentRequestsForFile(pFileObject);
    m_Parent->GetOutputPipe()->CancelSentRequestsForFile(pFileObject);
    return;
}
```

Код в листинге 9.14 довольно прямолинеен. Драйвер предоставляет вспомогательные функции `GetInputPipe` и `GetOutputPipe`, которые возвращают указатели интерфейса на получате-

лей ввода/вывода для его входного и выходного канала. С помощью этих указателей метод `OnCleanupFile` просто вызывает метод `IWDFIoTarget::CancelSentRequestsForFile` инфраструктуры, передавая ему указатель на интерфейс `IWDFFile` для объекта файла.

### Пример KMDF: отмена запросов ввода/вывода

Образец драйвера Osrusbf2 получает запросы от инфраструктуры через очередь и посыпает их каналу получателю USB. Когда устройство OSR USB Fx2 выходит из рабочего состояния или удаляется, инфраструктура вызывает функцию обратного вызова остановки очереди ввода/вывода по одному разу для каждого находящегося в транзите запроса, который драйвер получил из очереди. В зависимости от причины вызова, драйвер либо подтверждает обратный вызов для запроса, либо отменяет запрос.

В листинге 9.15 приводится пример подтверждения или отмены запроса в драйвере. Исходный код примера взят из файла Bulkwr.c.

#### Листинг 9.15. Отмена запроса драйвером KMDF

```
Void OsrFxEvtIoStop(IN WDFQUEUE Queue,
                     IN WDFREQUEST Request,
                     IN ULONG ActionFlags)
{
    . . . // Код опущен.

    if (ActionFlags == WdfRequestStopActionSuspend) {
        WdfRequestStopAcknowledge(Request, FALSE); // Не ставить
                                                    // в другую очередь.
    }
    else if (ActionFlags == WdfRequestStopActionPurge) {
        WdfRequestCancelSentRequest(Request);
    }
    return;
}
```

В листинге драйвер проверяет значение параметра `ActionFlags`, которое указывает причину вызова инфраструктурой функции обратного вызова `EvtIoStop`. Если устройство всего лишь покидает рабочее состояние, инфраструктура передает в нем значение `WdfRequestStopActionSuspend`. Драйвер подтверждает обратный вызов и оставляет запрос в транзите на получателе ввода/вывода. Но если инфраструктура передает значение `WdfRequestStopActionPurge`, указывающее удаление устройства, драйвер вызывает метод `WdfRequestCancelSentRequest`, чтобы отменить запрос.

## Получатели ввода/вывода FileHandle в драйверах UMDF

Драйвер UMDF взаимодействует с некоторыми устройствами с помощью файловых дескрипторов Windows. Примером таких устройств могут служить следующие:

- ◆ устройство, предоставляемое посредством сокета API Windows;
- ◆ устройство, эмулируемое с помощью именованных каналов Windows;
- ◆ устройство, эмулируемое с помощью файловой системы Windows.

Драйвер получает файловый дескриптор, вызывая функцию Windows (например, `socket`, `CreateFile` или `CreateNamedPipe`), которая возвращает файловый дескриптор.

Драйвер может присвоить (bind) этот дескриптор инфраструктурному получателю ввода/вывода, используя получатель ввода/вывода типа `FileHandle`. После этого драйвер может использовать инфраструктурный интерфейс получателя ввода/вывода, чтобы отправлять запросы ввода/вывода файловому дескриптору, таким образом пользуясь всеми преимуществами получателей ввода/вывода.

Интерфейс `IWDFFileHandleTargetFactory` создает получатель ввода/вывода, который ассоциируется с дескриптором файла. Драйвер запрашивает и получает указатель на этот интерфейс у инфраструктурного объекта устройства. После этого драйвер может вызвать метод `CreateFileHandleTarget`, передавая ему файловый дескриптор, для создания инфраструктурного объекта получателя.

Так как родителем объекта получателя является инфраструктурный объект устройства, то его время жизни по умолчанию такое же, как и родителя. Если драйвер больше не нуждается в объекте получателя ввода/вывода, в то время как объект устройства продолжает оставаться активным, драйвер может удалить объект получателя, вызывая метод `IWDFObject::DeleteWdfObject`.

В листинге 9.16 приведен пример создания драйвером UMDF получателя ввода/вывода `FileHandle`, представляющего именованный канал.

#### Листинг 9.16. Создание получателя ввода/вывода `FileHandle` в драйвере UMDF

```
HANDLE m_WriteHandle; // Дескриптор устройства
IWDFIoTarget * m_WriteTarget; // Получатель ввода/вывода
HRESULT hr = S_OK;
IWdffHandleTargetFactory * pFileHandleTargetFactory = NULL;
// Создаем канал и получаем дескриптор.
m_WriteHandle = CreateNamedPipe(NP_NAME,
    . . . // Код опущен, чтобы сохранить место.);
if (SUCCEEDED(hr)) {
    hr = m_FxDevice->QueryInterface(IID_PPV_ARGS(&pFileHandleTargetFactory));
    if (SUCCEEDED(hr)) {
        hr = pFileHandleTargetFactory->CreateFileHandleTarget(m_WriteHandle,
            &m_WriteTarget);
    }
}
. . . // Нерелевантный код опущен.
SAFE_RELEASE(pFileHandleTargetFactory);
```

В предшествующем листинге драйвер вызывает функцию Windows `CreateNamedPipe`, чтобы открыть дескриптор именованного канала. В случае успешного выполнения функции драйвер запрашивает у инфраструктурного объекта устройства указатель на интерфейс `IWDFFileHandleTargetFactory`. После этого он вызывает метод `CreateFileHandleTarget`, который создает инфраструктурный объект получателя ввода/вывода, соответствующий дескриптору файла, и возвращает указатель на интерфейс `IWDFIoTarget` объекта получателя ввода/вывода. Когда драйвер больше не использует интерфейс `IWDFFileHandleTargetFactory`, он освобождает свою ссылку на него.

После создания получателя ввода/вывода драйвер может вызывать методы интерфейсов `IWDFIoTarget` и `IWDFIoTargetStateManagement`, чтобы форматировать запросы ввода/вывода для

устройства получателя, извлекать информацию о получателе и управлять состоянием получателя.

### **Почему необходимо использовать получатель ввода/вывода *FileHandle* вместо Windows API?**

Если драйвер может создать дескриптор файла для получателя и драйвер не загружен, как часть стека устройств для узла devnode получателя, нужно использовать получатель ввода/вывода *FileHandle*. Например, если драйвер использует сокет, нужно использовать получатель ввода/вывода *FileHandle*. Но если драйвер загружается, как часть стека устройств для узла devnode получателя, нужно просто использовать стандартный получатель ввода/вывода.

Хотя можно непосредственно использовать Windows API, следует рассмотреть применение получателя ввода/вывода *FileHandle* по следующим причинам.

- Поток ввода/вывода проходит через все устройства в стеке устройства UMDF.
- Вы приобретаете все выгоды получателя ввода/вывода WDF: возможность управлять состоянием получателя, обрабатывать ввод/вывод в зависимости от состояния получателя, а также возможность отправлять ввод/вывод в различных конфигурациях, например, с тайм-аутом или без него или синхронно или асинхронно.
- Получатель ввода/вывода координирует завершение и отмену ввода/вывода, что может быть довольно трудной задачей, особенно если драйвер обрабатывает несколько незаконченных запросов ввода/вывода.

По этим же причинам нужно использовать получатель ввода/вывода для взаимодействия с частью режима ядра стека устройств. (*Правин Рао (Praveen Rao), команда разработчиков Windows Driver Foundation, Microsoft.*)

## **Получатели ввода/вывода USB**

Одной из основных задач в разработке WDF было сделать драйверную модель легко расширяемой, чтобы поддерживать новые типы аппаратного обеспечения. Как в UMDF, так и в KMDF первые специфичные для аппаратуры специализированные получатели ввода/вывода поддерживают устройства USB. Получатели ввода/вывода USB можно использовать для создания полностью функциональных драйверов для USB-устройств, использующих стек устройств USB Windows.

Хотя само использование устройств USB обычно не представляет проблем, программировать их может быть трудной задачей. Драйверы для устройств USB должны решать такие задачи, как неожиданное удаление устройства, управление его состояниями и очисткой, т. к. устройство USB может быть удалено в любой момент без какой-либо предварительной обработки. Хотя интерфейс драйверов USB сравнительно сложный, UMDF и KMDF представляют его абстракцию упорядоченным способом и упрощают многие стандартные операции, которые должны выполнять драйверы USB.

## **Устройства USB**

Данные между устройством USB и хостом USB передаются посредством абстракции, называемой *каналом* (pipe). Канал имеет конечную точку на устройстве, которая определяется адресом. На втором конце канала всегда находится хост-контроллер.

В терминологии USB направление передачи рассматривается по отношению к хосту. Таким образом, IN (входящий) всегда обозначает передачу от устройства хосту, а OUT (исходя-

щий) всегда обозначает передачу от хоста устройству. Устройства USB также поддерживают двунаправленную передачу управляющих данных. На рис. 9.3 показан хост USB и устройство с тремя конечными точками.

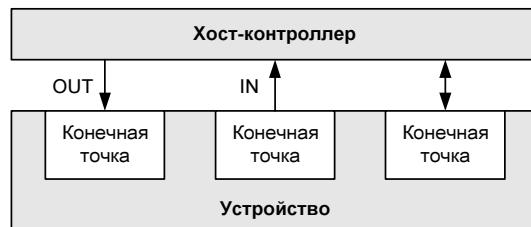


Рис. 9.3. Абстракция USB

Три конечные точки устройства USB на рис. 9.3 — это конечная точка OUT, конечная точка IN и конечная точка, применяемая для двунаправленных передач управляющих данных.

Конечные точки устройства группируются в функциональнее интерфейсы, набор которых составляет конфигурацию устройства. Например, устройство USB для резервного копирования единственным нажатием клавиши (one-touch backup device) может определять одну группу конечных точек как интерфейс HID, управляющий кнопкой этого самого единственного нажатия, и другую группу конечных точек как интерфейс, предоставляющий функцию массовой памяти для устройства. Конфигурация устройства состоит из этих двух интерфейсов.

Кроме этого, отдельный интерфейс может иметь несколько наборов настроек параметров. Возьмем, например, защитную заглушку (dongle) Bluetooth, в которой один интерфейс определен для управляющих данных и данных, не допускающих передачу с потерями (lossless data), а другой интерфейс — для голосовых данных, допускающих передачу с потерями (lossy data). Второй интерфейс имеет несколько альтернативных наборов настроек параметров, предоставляющих возрастающий уровень качества голоса и требующих все большую пропускную способность. Только один из альтернативных наборов настроек параметров может применяться в любой момент.

Конечные точки текущего набора настроек параметров ассоциируются с каналами и, поэтому, могут быть получателями ввода/вывода. Все прочие конечные точки являются просто конечными точками.

## Конфигурационные дескрипторы и дескрипторы устройств

Каждое устройство USB имеет дескриптор устройства, который предоставляет специфичную для устройства и поставщика информацию, например, версию спецификации USB, поддерживаемую устройством, класс устройства и имя устройства. Дескриптор устройства также содержит число конфигураций, поддерживаемых устройством. Но WDF и встроенные классы драйверов USB для Windows поддерживают только первую конфигурацию устройства.

Каждая конфигурация также имеет свой дескриптор. Дескриптор конфигурации описывает возможности энергопотребления и пробуждения данной конфигурации и содержит число интерфейсов в конфигурации. Конфигурация и ее интерфейсы следуют этим правилам.

- ◆ Конфигурация содержит один или несколько интерфейсов, все из которых активны одновременно.

- ◆ Каждый интерфейс имеет один или несколько наборов настроек параметров. Набор настроек параметров представляет собой коллекцию конечных точек. Каждый альтернативный набор настроек параметров в интерфейсе может иметь разное количество конечных точек или определенное количество конечных точек, потребляющих различные объемы пропускной способности шины.
- ◆ Конечная точка может находиться только в одном интерфейсе конфигурации, но может быть использована в нескольких альтернативных наборах в данном интерфейсе. Все конечные точки в альтернативном наборе настроек параметров могут использоваться одновременно. На рис. 9.4 показан пример конфигурации гипотетического устройства.

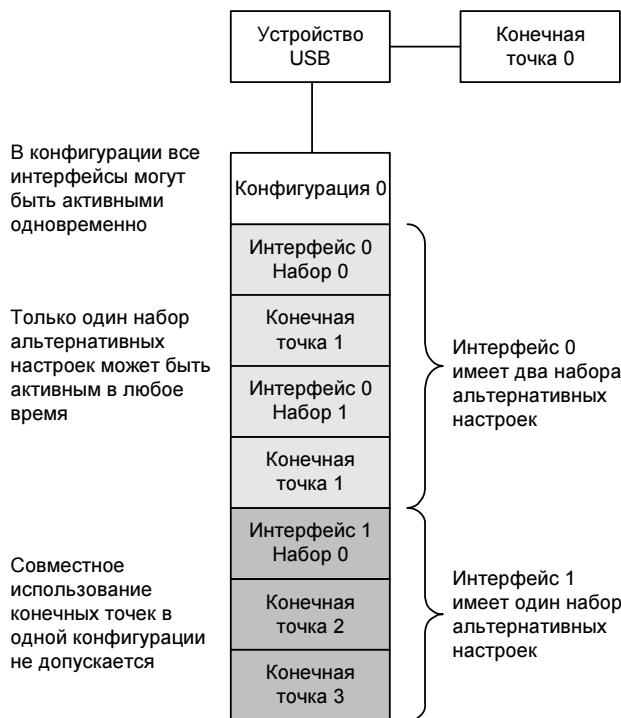


Рис. 9.4. Конфигурация USB-устройства

На рис. 9.4 интерфейс 0 имеет два набора альтернативных настроек, только один из которых может быть активным в любое время. Интерфейс 1 имеет только один набор альтернативных настроек.

При конфигурировании драйвер USB-устройства выбирает один или несколько интерфейсов и один из наборов альтернативных настроек для каждого интерфейса. Каждый интерфейс содержит одну или несколько конечных точек. Выбирая интерфейсы и набор установок в интерфейсе, драйвер указывает оказываемую им поддержку для определенных функций устройства.

Все устройства поддерживают конечную точку 0. Конечная точка 0 является управляющей конечной точкой, используемой для конфигурации интерфейсов.

Большинство устройств USB не предоставляют несколько интерфейсов или несколько альтернативных наборов настроек. Например, устройство OSR USB Fx2 имеет один интерфейс с одним набором настроек и три конечные точки.

## Модели передачи данных USB

USB поддерживает три модели ввода/вывода передачи данных — по прерыванию, групповую пересылку и изохронно — а также отдельную модель управляющего ввода/вывода. Все устройства USB поддерживают механизм управления, но поддержка механизмов передачи данных является необязательной. Для передачи данных используются односторонние конечные точки, а для управляющих передач — двунаправленные.

Характеристики моделей передач данных следующие.

- ◆ **Групповая пересылка (bulk transfers).** Односторонняя передача без гарантированной задержки (*guaranteed latency*) или полосы пропускания и гарантированной безошибочной доставкой.
- ◆ **Передача по прерыванию (interrupt transfer).** Односторонняя передача с гарантированной задержкой и повторной передачей при ошибке.
- ◆ **Изохронная передача (isochronous transfer).** Односторонняя передача с постоянной скоростью, гарантированной полосой пропускания и ограниченной задержкой (*bounded latency*), без повторной передачи при ошибках, что может вызвать потерю данных.

Каждая конечная точка ассоциируется с типом передачи данных. Конечные точки для передачи данных по прерыванию, групповой пересылкой и изохронно являются односторонними. Конечные точки для передачи управляющих данных являются двунаправленными, и обычно применяются для перечисления устройства USB и выбора рабочей конфигурации. Каждая конечная точка имеет уникальный адрес.

WDF поддерживает передачи по прерыванию и групповыми пересылками, а также стандартную конечную точку 0 для управляющей информации.

Для взаимодействия с изохронной конечной точкой драйвер KMDF может создать пакет URB, отформатировать этот пакет с помощью функций USBD, вставить его в объект запроса ввода/вывода с помощью методов WDF и, наконец, отправить пакет URB, используя метод `WdfRequestSend`. Для примера отправки пакета URB см. файл `Isorwr.c` драйвера `Usbsamp`.

### Почему KMDF не поддерживает изохронные конечные точки?

Хотя решение не предоставлять в KMDF собственной поддержки для изохронных конечных точек может казаться произвольным, в действительности оно было принято на основе двух факторов. Во-первых, мы провели исследования рынка устройств и обнаружили, что очень немногие устройства используют изохронные конечные точки, а для большинства устройств, применяющих их, поддержка предоставляется драйверами, поставляемыми с Windows. Во-вторых, у нас не было устройств, на которых можно было бы испытать наше решение. Обучающее устройство Fx2, несмотря на его небольшие размеры, является великолепным тестовым устройством, и во время разработки KMDF мы проверили на нем много возможностей USB и политики энергопотребления. Но мы получили его, когда уже зашли слишком далеко в процесс разработки KMDF, и, что имеет более уместное отношение к данному вопросу, в нем отсутствуют изохронные конечные точки.

Отсутствие полномасштабной поддержки не означает, что общие возможности объектов `WDFIOTARGET`, встроенные в объекты `WDFUSBPIPE`, не могут быть использованы для получения требуемого решения. Мы хотели быть уверенными, что для обработки изохронных данных можно было бы все-таки применять некоторые части KMDF. Например, изохронный пакет URB можно отформатировать с помощью метода `WdfUsbTargetPipeFormatRequest`.

ForUrb или WdfRequestWdmFormatUsingStackLocation, а отправить вниз по стеку — с помощью метода WdfRequestSend. Применение метода WdfRequestSend позволяет получить всю логику, необходимую для отслеживания пакетов и для остановок WDFIOTARGET, для изохронных передач данных. Можно также с легкостью использовать эти функции API для создания собственного непрерывного читателя для изохронной конечной точки. (Даун Холэн (Down Holan), команда разработчиков Windows Driver Foundation, Microsoft.)

## Специализированные получатели ввода/вывода USB в WDF

Драйверы WDF для устройств USB должны пользоваться поддержкой для специализированных получателей ввода/вывода USB, предоставляемой как в UMDF, так и в KMDF. В WDF определены следующие три типа объектов для применения с получателями ввода/вывода USB:

- ◆ *объект устройства получателя USB* представляет устройство USB и предоставляет методы для получения информации об устройстве и отправки запросов управления устройству;
- ◆ *объект интерфейса USB* представляет отдельный интерфейс и поддерживает методы, с помощью которых драйвер может выбрать альтернативный набор настроек и получить информацию о настройках;
- ◆ *объект канала получателя USB* представляет отдельный канал, т. е. конечную точку, сконфигурированную в текущем наборе установок интерфейса.

Интерфейсы и методы, поддерживающие USB, приведены в табл. 9.10.

**Таблица 9.10. Интерфейсы и методы для поддержки USB**

Абстракция USB	Интерфейс UMDF	Тип объекта и метод KMDF
Устройство USB	IWDFUsbTargetDevice	WDFUSBDEVICE WdfUsbTargetDeviceXxx
Интерфейс	IWDFUsbInterface	WDFUSBINTERFACE WdfUsbInterfaceXxx
Канал	IWDFUsbTargetPipe	WDFUSBPIPE WdfUsbTargetPipeXxx

Драйвер создает объекты устройства получателя, интерфейса и канала получателя при конфигурировании USB, которое обычно выполняется в функции обратного вызова драйвера для подготовки аппаратной части. Соответственно:

- ◆ функциональный драйвер UMDF для устройства USB должен реализовывать интерфейс IPnpCallbackHardware на объекте обратного вызова устройства;
- ◆ драйвер KMDF должен зарегистрировать функции обратного вызова EvtDeviceDOEntry, EvtDeviceDOExit и EvtPrepareHardware.

### Внимание!

Драйвер UMDF, использующий получатель ввода/вывода USB, должен указывать WinUSB в качестве диспетчера ввода/вывода в INF-файле, как было описано в разд. "Диспетчеры ввода/вывода UMDF" ранее в этой главе.

## Объекты устройств исполнителей USB

Драйвер вызывает инфраструктуру для создания объекта устройства получателя USB следующим образом.

- ◆ Драйвер UMDF запрашивает интерфейс `IWDFUsbTargetFactory` на объекте устройства, после чего использует полученный указатель для вызова метода `CreateUsbTargetDevice`. Этот метод создает инфраструктурный объект устройства получателя USB и возвращает указатель на интерфейс `IWDFUsbTargetDevice`.

Созданный инфраструктурой объект устройства получателя USB уже сконфигурированный установками по умолчанию и альтернативным набором настроек 0 для каждого интерфейса.

- ◆ Драйвер KMDF вызывает метод `WdfUsbTargetDeviceCreate`, который возвращает дескриптор объекта типа `WDFUSBDEVICE`. В отличие от большинства других методов KMDF для создания объектов, этот метод не требует структуры конфигурации объекта. После создания устройства получателя USB драйвер KMDF должен сконфигурировать его, что он и делает, вызывая метод `WdfUsbTargetDeviceSelectConfig`.

По умолчанию объект исполнителя ввода/вывода USB является потомком объекта устройства.

После создания объекта устройства исполнителя USB драйвер вызывает методы для получения информации о конфигурации и дескрипторы устройства. WDF также поддерживает методы для выполнения следующих типов специфичных для устройства запросов для получателя ввода/вывода устройства USB:

- ◆ форматирование и отправка запросов IOCTL каналу управления;
- ◆ извлечение различной другой информации об устройстве;
- ◆ сброс и последующее включение питание порта (только в KMDF);
- ◆ форматирование и отправка пакетов URB WDM (только в KMDF).

Драйвер KMDF может использовать методы `WdfIoTargetStart` и `WdfIoTargetStop` для управления получателями ввода/вывода USB, как для любых других получателей ввода/вывода. Когда драйвер KMDF указывает дескриптор объекта типа `WDFUSBDEVICE` при вызове одного из этих методов, инфраструктура запускает или останавливает все сконфигурированные в настоящий момент каналы на всех интерфейсах для объекта устройства USB. Например, если драйвер вызывает метод `WdfIoTargetStop` с дескриптором объекта типа `WDFUSBDEVICE`, кроме прекращения ввода/вывода на само устройство получателя ввода/вывода USB, инфраструктура также прекращает отправку запросов ввода/вывода в каналы, являющихся частью каждого интерфейса. Соответственно, драйверу не требуется обходить в цикле все каналы на всех интерфейсах, чтобы остановить или запустить каждый из них по отдельности.

## Объекты интерфейса USB

При выполнении драйвером конфигурирования устройства инфраструктура создает объект интерфейса USB для каждого интерфейса в конфигурации. По умолчанию объект интерфейса USB является потомком объекта устройства. Чтобы получить доступ к объектам интерфейса, драйвер вызывает метод на объекте устройства получателя ввода/вывода USB следующим образом.

- ◆ Драйвер UMDF вызывает метод `IWDFUsbTargetDevice::RetrieveUsbInterface`, передавая ему номер интерфейса, чтобы получить указатель на интерфейс `IWDFUsbInterface` для данного USB-интерфейса объекта.

- ◆ Драйвер KMDF вызывает метод `WdfUsbTargetDeviceGetInterface`, передавая ему номер интерфейса, чтобы получить дескриптор объекта типа `WDFUSBINTERFACE` для данного интерфейса.

Выбрав интерфейс, драйвер может выбрать альтернативный набор настроек в этом интерфейсе, после чего он может извлечь информацию о каналах в этих настройках. По умолчанию в каждом интерфейсе инфраструктура использует альтернативный набор настроек 0.

## Объекты каналов исполнителей USB

Канал представляет собой конечную точку, являющуюся частью текущего альтернативного набора интерфейса. Инфраструктура создает объект канала для каждого канала в наборе настроек. Драйвер получает доступ к каналам следующим образом:

- ◆ драйвер UMDF вызывает метод `IWDFUsbInterface::RetrieveUsbPipeObject` на инфраструктурном объекте интерфейса USB, чтобы получить указатель на интерфейс `IWDFUsbTargetPipe` на определенном канале;
- ◆ драйвер KMDF вызывает метод `WdfUsbInterfaceGetConfiguredPipe`, чтобы получить дескриптор объекта типа `WDFUSBPIPE` для данного канала.

Для каналов WDF поддерживает методы, возвращающие информацию о конфигурации канала, управляющие вводом/выводом на канале и управляющие политикой канала, например, лимитами на размер пакета.

Время жизни объекта связано со временем жизни текущих настроек интерфейса. KMDF удаляет объекты каналов явным образом, когда драйвер выбирает новый набор альтернативных настроек. Для удаления неиспользуемых объектов UMDF полагается на подсчет ссылок и объектную модель WDF.

## Конфигурирование получателей ввода/вывода USB

Прежде чем устройство может принимать любые запросы ввода/вывода, за исключением запросов, направленных в канал управления, функциональный драйвер устройства USB должен сконфигурировать его. В зависимости от конструкции устройства, процесс конфигурирования может состоять из выполнения следующих операций:

- ◆ получение информации о текущей конфигурации, например, о количестве интерфейсов;
- ◆ извлечение объектов интерфейса;
- ◆ если интерфейс поддерживает несколько наборов настроек, выбор альтернативного набора настроек;
- ◆ извлечение каналов в каждом интерфейсе.

Если устройство имеет только один интерфейс и один набор настроек, большинство этих шагов можно пропустить и просто извлечь каналы.

В следующем примере показано использование обучающего устройства OSR Fx2, сконфигурированного так:

- ◆ число конфигураций — 1;
- ◆ число интерфейсов — 1;
- ◆ число наборов настроек — 1;
- ◆ число конечных точек — 3;

◆ типы передач данных и их направления:

- по прерыванию — IN (входящая);
- групповая пересылка — OUT (выходящая);
- групповая пересылка — IN (входящая).

## Пример UMDF: конфигурирование получателя ввода/вывода USB

Драйвер UMDF создает инфраструктурный объект получателя USB и конфигурирует устройство получателя USB в методе `OnPrepareHardware` интерфейса `IPnpCallbackHardware` объекта устройства. Весь код, представленный в этом разделе, взят из файла `Device.cpp` образца драйвера `Fx2_Driver`, но был слегка отредактирован для целей примера.

Первой задачей драйвера является создать инфраструктурный объект устройства получателя для устройства USB. Код для выполнения этой задачи приводится в листинге 9.17. В сущности, объект устройства подключает драйвер к подсистеме USB.

### Листинг 9.17. Создание объекта устройства исполнителя USB в драйвере UMDF

```

HRESULT hr;
IWDFUsbTargetFactory * pIUsbTargetFactory = NULL;
IWDFUsbTargetDevice * pIUsbTargetDevice = NULL;
ULONG length;
UCHAR m_Speed;
hr = m_FxDevice->QueryInterface (IID_PPV_ARGS(&pIUsbTargetFactory));
if (FAILED(hr)) {
    . . . // Код для обработки ошибок опущен.
}
if (SUCCEEDED(hr)) {
    hr = pIUsbTargetFactory->CreateUsbTargetDevice(&pIUsbTargetDevice);
    length = sizeof(UCHAR);
    hr = pIUsbTargetDevice->RetrieveDeviceInformation(DEVICE_SPEED,
                                                       &length,
                                                       &m_Speed);
}

```

Для создания в инфраструктуре объекта устройства получателя USB драйвер использует интерфейс `IWDFUsbTargetFactory`. Он запрашивает этот интерфейс на объекте устройства, после чего использует полученный указатель для вызова метода `CreateUsbTargetDevice` для создания инфраструктурного объекта устройства. Метод `CreateUsbTargetDevice` возвращает указатель на интерфейс `IWDFUsbTargetDevice`. Теперь драйвер может вызвать метод `RetrieveDeviceInformation`, чтобы получить скорость устройства.

Дальше драйвер определяет количество интерфейсов USB в устройстве и извлекает указатель на интерфейс. Соответствующий код приведен в листинге 9.18.

### Листинг 9.18. Извлечение интерфейса USB в драйвере UMDF

```

IWDFUsbInterface * pIUsbInterface = NULL;
UCHAR NumEndPoints = 0;
UCHAR NumInterfaces = pIUsbTargetDevice->GetNumInterfaces();
WUDF_TEST_DRIVER_ASSERT(1 == NumInterfaces);

```

```
hr = pIUsbTargetDevice->RetrieveUsbInterface(0, &pIUsbInterface);
if (FAILED(hr)) {
    . . . // Код для обработки ошибок опущен.
}
NumEndPoints = pIUsbInterface->GetNumEndPoints();
if (NumEndPoints != NUM_OSRUSB_ENDPOINTS) {
    hr = E_UNEXPECTED;
}
```

Чтобы узнать количество интерфейсов в устройстве, драйвер вызывает метод `IWDFUsbTargetDevice::GetNumInterfaces`, который возвращает количество интерфейсов в конфигурации по умолчанию. Данный драйвер полагает, что устройство имеет один интерфейс и объявляет ошибку в противном случае. Интерфейсам присвоены номера, начиная с нуля; поэтому, при вызове драйвером метода `IWDFUsbTargetDevice::RetrieveUsbInterface`, он передает ему 0 в первом параметре, указывая инфраструктуре возвратить указатель `IWDFUsbInterface` для первого интерфейса. Дальше драйвер может получить количество конечных точек в интерфейсе, вызывая для этого метод `IWDFUsbInterface::GetNumEndPoints`.

На данном этапе драйвер имеет необходимую информацию для выполнения конфигурирования каналов. Вспомните, что канал представляет собой конечную точку, используемую в текущем наборе альтернативных настроек интерфейса. Код для выполнения конфигурирования каналов приведен в листинге 9.19.

#### Листинг 9.19. Выполнение конфигурирования каналов USB в драйвере UMDF

```
IWDFUsbTargetPipe * pIUsbPipe = NULL;
IWDFUsbTargetPipe * pIUsbInputPipe = NULL;
IWDFUsbTargetPipe * pIUsbOutputPipe = NULL;
IWDFUsbTargetPipe * pIUsbInterruptPipe = NULL;
for (UCHAR PipeIndex = 0; PipeIndex < NumEndPoints; PipeIndex++) {
    hr = pIUsbInterface->RetrieveUsbPipeObject(PipeIndex, &pIUsbPipe);
    if (FAILED(hr)) {
        . . . // Код для обработки ошибок опущен.
    }
    else {
        if (pIUsbPipe->IsInEndPoint()) {
            if (UsbdPipeTypeInterrupt == pIUsbPipe->GetType() ) {
                pIUsbInterruptPipe = pIUsbPipe;
            }
            else if (UsbdPipeTypeBulk == pIUsbPipe->GetType() ) {
                pIUsbInputPipe = pIUsbPipe;
            }
            else {
                SAFE_RELEASE(pIUsbPipe);
            }
        }
        else if (pIUsbPipe->IsOutEndPoint()
                  && (UsbdPipeTypeBulk == pIUsbPipe->GetType() ) ) {
            pIUsbOutputPipe = pIUsbPipe;
        }
        else {
            SAFE_RELEASE(pIUsbPipe);
        }
    }
}
```

```
if (NULL == pIUsbInputPipe || NULL == pIUsbOutputPipe) +  
    hr = E_UNEXPECTED;  
}
```

Номера конечных точек, которые также называются индексами, начинаются с нуля. Как можно видеть в листинге 9.19, драйвер проходит в цикле по всем конечным точкам, извлекая указатель на интерфейс IWDFUsbTargetPipe для соответствующего канала, после чего определяет следующую информацию для каждого канала:

- ◆ является ли данный канал входным или выходным;
  - ◆ поддерживает ли канал передачи по прерыванию или групповые передачи.

Драйвер извлекает указатель на интерфейс IWDFUsbTargetPipe, вызывая метод IWDFUsbInterface::RetrieveUsbPipeObject, и использует возвращенный указатель для вызова методов IWDFUsbTargetPipe::IsInEndPoint, IsOutEndPoint и GetType. Так как драйвер разработан для обучающего устройства OSR USB Fx2, он ожидает найти входящий канал (IN) для передач по прерыванию, входящий канал (IN) для групповых передач и выходящий (OUT) канал для групповых передач.

Метод `IsInEndPoint` возвращает `TRUE` для каналов `IN`, и драйвер вызывает метод `GetType`, чтобы определить, какой тип передач данных поддерживается данным каналом. Метод `GetType` возвращает одно из следующих значений перечисления `USBPIPE TYPE`:

- ◆ UsbdPipeTypeControl;
  - ◆ UsbdPipeTypeIsochronous;
  - ◆ UsbdPipeTypeBulk;
  - ◆ UsbdPipeTypeInterrupt.

Если канал поддерживает передачи по прерыванию или групповые передачи, драйвер сохраняет указатель на интерфейс канала получателя как `pIUsbInterruptPipe` или `pIUsbInputPipe` соответственно.

Если же данный канал является выходящим (OUT) для групповых передач, то драйвер сохраняет указатель на интерфейс канала получателя как `pIUsbOutputPipe`.

Если по окончанию цикла драйвер не нашел ожидаемых каналов, то он устанавливает код ошибки. Теперь драйвер может выполнить конфигурирование каналов. Исходный код для этой операции показан в листинге 9.20.

**Листинг 9.20.** Выполнение конфигурирования каналов USB в драйвере UMDF

```
if (FAILED(hr)) {  
    . . . // Код для обработки ошибок опущен.  
}
```

UMDF поддерживает настройки политики канала для управления многими аспектами работы устройства, включая, среди прочих, значения тайм-аутов и то, каким образом устройство реагирует на приостановление передачи данных. Константы, идентифицирующие типы политик, определены в заголовочном файле Winusbio.h механизма WinUSB.

Дополнительную информацию о политике каналов см. в разделе **WinUsb\_SetPipePolicy** в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80619>.

Образец драйвера Fx2\_Driver присваивает константу `ENDPOINT_TIMEOUT` для значений тайм-аута для входного и выходного канала. Эта константа определена как 10 000 (10 секунд) в заголовочном файле Device.h. Механизм WinUSB отменяет передачи, которые не завершаются в течение указанного периода тайм-аута.

## Пример KMDF: конфигурирование получателя ввода/вывода USB

Драйвер KMDF создает и конфигурирует объект устройства получателя ввода/вывода USB в своей функции обратного вызова `EvtDevicePrepareHardware`. Код примера в этом разделе основан на коде из файла Decive.c образца драйвера Osrusbf2.

Чтобы создать объект устройства получателя ввода/вывода USB, драйвер KMDF вызывает метод `WdfUsbTargetDeviceCreate`, как показано в листинге 9.21.

### Листинг 9.21. Создание объекта устройства получателя USB в драйвере KMDF

```
NTSTATUS status;  
PDEVICE_CONTEXT pDeviceContext;  
WDF_USB_DEVICE_INFORMATION deviceInfo;  
pDeviceContext = GetDeviceContext(Device);  
status = WdfUsbTargetDeviceCreate(Device,  
                                    WDF_NO_OBJECT_ATTRIBUTES,  
                                    &pDeviceContext->UsbDevice);
```

В качестве параметров методу `WdfUsbTargetDeviceCreate` передаются дескриптор объекта устройства и указатель на структуру `WDF_OBJECT_ATTRIBUTES`; метод возвращает дескриптор объекта типа `WDFUSBDEVICE`.

### Выбор конфигурации

Если инфраструктура успешно создает объект устройства USB, драйвер выбирает конфигурацию устройства и извлекает информацию из дескриптора конфигурации устройства, вызывая для этого метод `WdfUsbTargetDeviceSelectConfig`. Этот метод выполняет конфигурирование устройства, создает объекты WDF для каналов и интерфейса USB и возвращает информацию об указанной конфигурации.

В качестве входного и выходного параметров метод `WdfUsbTargetDeviceSelectConfig` требует структуру `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS`. При вызове метода структура `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS` указывает конфигурацию. По возвращению метода структура содержит информацию о выбранной конфигурации из дескриптора конфигурации устройства.

Дескриптор конфигурации устройства содержит информацию о многочисленных аспектах устройства — его конфигурациях, их интерфейсах и т. д. KMDF позволяет драйверу большую гибкость в выполнении конфигурирования устройства. Соответственно, инфраструктура предоставляет несколько функций семейства `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_XXX` для выполнения инициализации устройства. Отдельные функции этого семейства приводятся в табл. 9.11.

**Таблица 9.11. Функции инициализации для структуры `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS`**

Функция	Описание
DECONFIG	Обнуляет структуру конфигурации, таким образом указывая, что не был выбран ни один интерфейс
INTERFACES_DESCRIPTORS	Конфигурирует устройство, указывает дескриптор конфигурации и массив дескрипторов интерфейсов
MULTIPLE_INTERFACE	Конфигурирует устройство для использования нескольких интерфейсов
SINGLE_INTERFACE	Конфигурирует устройство для использования одного интерфейса. Этую функцию могут использовать драйверы для большинства устройств
URB	Конфигурирует устройство, указывая пакет URB

### Примечание

KMDF предоставляет дополнительные методы, которые драйвер может вызывать, чтобы получить специфическую информацию о USB-устройстве, прежде чем выполнять его конфигурирование. Одними из таких методов, среди прочих, являются методы `WdfUsbTargetDeviceRetrieveConfigDescriptor`, `WdfUsbTargetDeviceGetDeviceDescriptor` и `WdfUsbTargetDeviceGetInterface`.

В листинге 9.22 показан пример выбора конфигурации драйвером и получение информации об этой конфигурации.

#### Листинг 9.22. Выбор конфигурации устройства USB в драйвере KMDF

```
WDF_USB_DEVICE_SELECT_CONFIG_PARAMS configParams;
NTSTATUS status;
PDEVICE_CONTEXT pDeviceContext;
UCHAR numberConfiguredPipes;

pDeviceContext = GetDeviceContext(Device);
WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_SINGLE_INTERFACE(&configParams);
status = WdfUsbTargetDeviceSelectConfig(pDeviceContext->UsbDevice,
                                         WDF_NO_OBJECT_ATTRIBUTES,
                                         &configParams);

if (!NT_SUCCESS(status)) return status;
pDeviceContext->UsbInterface =
    configParams.Types.SingleInterface.ConfiguredUsbInterface;
numberConfiguredPipes =
    configParams.Types.SingleInterface.NumberConfiguredPipes;
```

Образец драйвера Osrusbf2 выбирает первый интерфейс в первом дескрипторе конфигурации USB-устройства, вызывая метод `WdfUsbTargetDeviceSelectConfig` и передавая ему структуру `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS`.

Драйвер инициализирует структуру с помощью функции `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_SINGLE_INTERFACE`. Эта функция указывает, что устройство имеет лишь один интерфейс USB.

Метод `WdfUsbTargetDeviceSelectConfig` возвращает дескриптор для выбранного интерфейса в поле `Types.SingleInterface.ConfiguredUsbInterface` структуры `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS`. Драйвер сохраняет этот дескриптор в поле `UsbInterface` области контекста устройства.

Образец драйвера OSR USB Fx2 имеет лишь один интерфейс с одним набором настроек, поэтому драйверу не требуется выбирать набор настроек.

### Перечисление каналов

Каждый интерфейс ассоциируется с одним или несколькими наборами альтернативных настроек и каждый набор альтернативных настроек ассоциируется с одной или несколькими конечными точками. Каждая конечная точка в выбранном наборе настроек представляет собой односторонний канал для определенного типа передачи данных. Метод `WdfUsbTargetDeviceSelectConfig` создает объект типа `WDFUSBPIPE` для каждого канала в интерфейсе и возвращает количество сконфигурированных каналов в поле `Types.SingleInterface.NumberConfiguredPipes` структуры `WDF_USB_DEVICE_SELECT_CONFIG_PARAMS`. Драйвер сохраняет это значение в локальной переменной `numberConfiguredPipes`.

Дальше образец драйвера перечисляет все дескрипторы каналов USB, ассоциированные с выбранным интерфейсом, как показано в листинге 9.23. Этот шаг не является абсолютно необходимым, но он служит для демонстрации получения доступа к коллекции `WDFUSBPIPE`, ассоциированной с интерфейсом USB.

#### Листинг 9.23. Перечисление каналов для интерфейса USB в драйвере KMDF

```
WDFUSBPIPE                                pipe;
WDF_USB_PIPE_INFORMATION                  pipeInfo;
UCHAR                                     index;
for (index=0; index < numberConfiguredPipes; index++) {
    WDF_USB_PIPE_INFORMATION_INIT(&pipeInfo);
    pipe = WdfUsbInterfaceGetConfiguredPipe
        (pDeviceContext->UsbInterface,
         index, //PipeIndex,
         &pipeInfo);

    // Информируем инфраструктуру, что можно считывать
    // меньше чем MaximumPacketSize.
    WdfUsbTargetPipeSetNoMaximumPacketSizeCheck(pipe);
    if (WdfUsbPipeTypeInterrupt == pipeInfo.PipeType) {
        pDeviceContext->InterruptPipe = pipe;
    }
    if (WdfUsbPipeTypeBulk == pipeInfo.PipeType
        && WdfUsbTargetPipeIsInEndpoint(pipe)) {
        pDeviceContext->BulkReadPipe = pipe;
    }
    if (WdfUsbPipeTypeBulk == pipeInfo.PipeType
        && WdfUsbTargetPipeIsOutEndpoint(pipe)) {
        pDeviceContext->BulkWritePipe = pipe;
    }
}
```

```
// Если не найдены все три канала, запуск завершается неудачей.
if (!(pDeviceContext->BulkWritePipe && pDeviceContext->BulkReadPipe
    && pDeviceContext->InterruptPipe)) {
    status = STATUS_INVALID_DEVICE_STATE;
    return status;
}
```

Методу требуется передать следующие три параметра:

- ◆ дескриптор интерфейса USB, указывающий интерфейс, содержащий канал;
- ◆ индекс канала (с отсчетом от нуля);
- ◆ необязательный выходной параметр, указывающий на расположение хранилища для структуры `WDF_USB_PIPE_INFORMATION`.

Для каждого дескриптора USB-интерфейса инфраструктура содержит коллекцию сконфигурированных каналов в текущем наборе. Значение переменной, передаваемое во втором параметре, указывает индекс (с отсчетом от нуля) канала, о котором необходимо получить информацию.

Необязательный третий, выходной, параметр, указывает расположение хранилища для структуры `WDF_USB_PIPE_INFORMATION`. Если драйвер предоставляет структуру, инфраструктура заполняет ее информацией об указанном канале. Эта функция инициализирует структуру `WDF_USB_PIPE_INFORMATION` и передает адрес этой структуры в третьем параметре метода `WdfUsbInterfaceGetConfiguredPipe`.

Драйвер проходит в цикле через всю коллекцию до тех пор, пока он не извлечет информацию обо всех каналах и убедится в том, что каналы соответствуют конфигурации устройства, которую драйвер ожидал обнаружить.

По умолчанию инфраструктура сообщает об ошибке, если драйвер использует буфер для чтения, размер которого не является кратным целому числу максимальных размеров пакета канала. Эта проверка размера буфера помогает предотвратить получение драйвером бесмысленных данных (*babble*), сгенерированных неожиданной деятельностью нашине. В цикле драйвер отключает эту проверку, вызывая метод `WdfUsbTargetPipeSetNoMaximumPacketSizeCheck` для каждого канала.

Кроме коллекции сконфигурированных каналов, инфраструктура содержит коллекцию альтернативных наборов настроек для каждого интерфейса. Каждый набор альтернативных настроек представляет собой коллекцию конечных точек. Драйвер может получить информацию об этой коллекции и содержащихся в ней конечных точках, вызывая методы `WdfUsbInterfaceGetNumEndpoints` и `WdfUsbInterfaceGetEndpointInformation` до или после выполнения конфигурирования устройства.

### **Получение информации об особенностях устройства**

Выполнив конфигурирование каналов, драйвер Osrusbf2 вызывает инфраструктуру, чтобы получить дополнительную информацию об устройстве (листинг 9.24).

#### **Листинг 9.24. Получение информации об устройстве USB**

```
WDF_USB_DEVICE_INFORMATION deviceInfo;
ULONGC waitWakeEnable;
WDF_USB_DEVICE_INFORMATION_INIT(&deviceInfo);
```

```
status = WdfUsbTargetDeviceRetrieveInformation
        (pDeviceContext->UsbDevice,
         &deviceInfo);

waitWakeEnable = deviceInfo.Traits
    & WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE;
if (waitWakeEnable){
    status = OsrFxSetPowerPolicy(Device);
    if (!NT_SUCCESS (status)) return status;
}
```

В листинге 9.24 драйвер инициализирует структуру `WDF_USB_DEVICE_INFORMATION` и передает ее методу `WdfUsbTargetDeviceRetrieveInformation`, чтобы получить информацию об устройстве. Метод возвращает эту структуру с информацией о версии USB, поддерживаемой устройством и его драйвером HCD (Host Controller Driver, драйвер контроллера хоста), возможностях драйвера HCD и набор флагов, указывающий, имеет ли устройство собственный источник питания, поддерживает ли оно пробуждение по сети и является ли оно высокоскоростным устройством.

Если устройство поддерживает пробуждение по сети, драйвер разрешает эту возможность в своих настройках политики энергопотребления. Драйвер проверяет значение бита пробуждения, возвращенного в поле `traits` структуры, и, если требуется, вызывает вспомогательную функцию для установки политики энергопотребления устройства.

## Отправление запроса ввода/вывода получателю USB

Чтобы отправить запрос ввода/вывода получателю USB, драйвер выполняет ту же самую последовательность шагов, что и для любого другого получателя ввода/вывода. А именно:

1. Создает запрос или использует запрос, доставленный инфраструктурой.
2. Организовывает объекты памяти и буферы для запроса.
3. Форматирует запрос.
4. Если требуется, регистрирует для запроса обратный вызов завершения ввода/вывода.
5. Отправляет запрос.

Инфраструктура WDF предоставляет специфичные для USB методы для форматирования запросов, для отправления определенных типов запросов и для получения параметров завершения.

### Пример UMDF: отправление синхронного запроса получателю ввода/вывода USB

Чтобы отправить запрос IOCTL получателю ввода/вывода USB, UMDF-драйвер использует метод `IWDFIoRequest::Send`, точно так же, как и для отправления запросов получателям ввода/вывода любого другого типа. Разница заключается в том, что в данном случае для форматирования запроса драйвер использует специфичные для USB интерфейсы UMDF.

Запросы для получателей ввода/вывода USB форматируются с помощью метода объекта получателя ввода/вывода USB `IWDFUsbTargetDevice::FormatRequestForControlTransfer`. В качестве параметров этому методу передается указатель на интерфейс `IWDFIoRequest` для запроса, указатель на структуру `WINUSB_SETUP_PACKET`, указатель на интерфейс `IWDFMemory` объекта памяти, содержащего буфер запроса, и необязательное значение смещения для буфера.

Структура `WINUSB_SETUP_PACKET` определена в заголовочном файле `Winusbio.h` и включена в WDK. Драйвер заполняет структуру `WINUSB_SETUP_PACKET` информацией о запросе, после чего вызывает метод `FormatRequestForControlTransfer`, чтобы отформатировать запрос как запрос IOCTL. В случае успешного выполнения метода, образец драйвера вызывает метод `IWDFIoRequest::Send`, чтобы отправить запрос получателю ввода/вывода USB.

В листинге 9.25 приведен пример форматирования и отправления запроса образцом драйвера `Fx2_Driver`.

#### Листинг 9.25. Отправление запроса IOCTL устройству USB драйвером UMDF

```
hr = m_pIUsbTargetDevice->FormatRequestForControlTransfer
    (pWdfRequest,
     SetupPacket,
     FxMemory,
     NULL //Transfer-Offset);
}

if (SUCCEEDED(hr)) {
    hr = pWdfRequest->Send (m_pIUsbTargetDevice,
                           WDF_REQUEST_SEND_OPTION_SYNCHRONOUS,
                           0); // Тайм-аут
}
```

Драйвер уже создал объект запроса ввода/вывода и объект памяти для запроса и сохранил указатели на их интерфейсы `IWDFIoRequest` и `IWDFMemory` в `pWdfRequest` и `FxMemory` соответственно. Методу `FormatRequestForControlTransfer` необходимо передать эти два указателя, а также указатель на структуру `WINUSB_SETUP_PACKET`. Для параметра смещения буфера драйвер передает значение `NULL`, указывая, что передача начинается с начала буфера, описываемого объектом памяти.

В случае успешного форматирования запроса инфраструктурой драйвер вызывает метод `IWDFIoRequest::Send`, чтобы отправить запрос получателю ввода/вывода USB, указывая при этом флаг синхронного запроса.

По завершению запроса драйвер получает статус завершения и специфичную для USB информацию завершения, как показано в листинге 9.26.

#### Листинг 9.26. Получение результатов запроса ввода/вывода USB в драйвере UMDF

```
*LengthTransferred = 0;
IWDFRequestCompletionParams * FxComplParams = NULL;
IWDFUsbRequestCompletionParams * FxUsbComplParams = NULL;
pWdfRequest->GetCompletionParams (&FxComplParams);
hr = FxComplParams->GetCompletionStatus();
if (SUCCEEDED(hr)) {
    HRESULT hrQI =
        FxComplParams->QueryInterface (IID_PPV_ARCS (&FxUsbComplParams));
    FxUsbComplParams->GetDeviceControlTransferParameters
        (NULL,
         LengthTransferred,
         NULL,
         NULL);
}
```

```
SAFE_RELEASE(FxUsbComplParams);
SAFE_RELEASE(FxComplParams);
```

В представленном листинге драйвер вызывает метод `IWDFIoRequest::GetCompletionParams` на объекте запроса, чтобы получить указатель на интерфейс `IWDFRequestCompletionParams` для объекта параметров завершения. Он передает этот указатель методу `IWDFRequestCompletionParams::GetCompletionStatus`, который он вызывает, чтобы получить статус завершения запроса.

В случае успешного завершения запроса драйвер запрашивает интерфейс `IWDFUsbRequestCompletionParams`, который поддерживает методы для получения специфичных для USB данных завершения. Метод `GetDeviceControlTransferParameters` возвращает указатель на интерфейс `IWDFMemory` для буфера вывода, количество переданных байтов, смещение в буфере вывода и указатель на настроочный пакет запроса. Наш драйвер заинтересован только в количестве переданных байтов, поэтому он вызывает этот метод с указателем на переменную для получения этого значения и значениями `NULL` для всех остальных параметров. Потом он освобождает указатели интерфейса для запроса и параметров USB.

### **Пример KMDF: отправление асинхронного запроса ввода/вывода получателю USB**

Для отправления запроса получателю ввода/вывода USB драйвер KMDF применяет методы, приведенные в табл. 9.12.

**Таблица 9.12. Методы для отправления запросов получателю ввода/вывода USB**

Тип запроса	Применяемый метод
Сброс и последующее включение питания порта (асинхронный)	<code>WdfUsbTargetDeviceFormatRequestForCyclePort</code> и <code>x4WdfRequestSend</code>
Сброс и последующее включение питания порта (синхронный)	<code>WdfUsbTargetDeviceCyclePortSynchronously</code>
Запрос IOCTL (асинхронный)	<code>WdfUsbTargetDeviceFormatRequestForControlTransfer</code> и <code>x4WdfRequestSend</code>
Запрос IOCTL (синхронный)	<code>WdfUsbTargetDeviceSendControlTransferSynchronously</code>
Получить строковый дескриптор (синхронный или асинхронный)	<code>WdfUsbTargetDeviceFormatRequestForString</code> и <code>X5WdfRequestSend</code>
Выполнить сброс порта (только синхронный)	<code>WdfUsbTargetDeviceResetPortSynchronously</code>
Пакет URB (асинхронный)	<code>WdfUsbTargetDeviceFormatRequestForRead</code> и <code>X5WdfRequestSend</code>
Пакет URB (синхронный)	<code>WdfUsbTargetDeviceSendUrbSynchronously</code>

Если драйвер применяет метод `WdfRequestSend`, прежде чем отправить запрос, он должен его отформатировать методом семейства `WdfUsbTargetDeviceFormatXxx` или семейства `WdfUsbTargetPipeFormatXxx` для устройства USB получателя или канала соответственно.

Методы, применяемые драйверами KMDF для отправления запросов каналу получателя USB, приведены в табл. 9.13.

**Таблица 9.13. Методы для отправления запросов каналу получателю ввода/вывода USB**

Тип запроса	Применимый метод
Отмена (синхронный)	WdfUsbTargetPipeAbortSynchronously
Отмена (асинхронный)	WdfUsbTargetPipeFormatRequestForAbort и X5WdfRequestSend
На чтение (асинхронный)	WdfUsbTargetPipeFormatRequestForRead и X5WdfRequestSend
На чтение (синхронный)	WdfUsbTargetPipeReadSynchronously
Сброс (асинхронный)	WdfUsbTargetPipeFormatRequestForReset и X5WdfRequestSend
Сброс (синхронный)	WdfUsbTargetPipeResetSynchronously
Пакет URB (асинхронный)	WdfUsbTargetPipeFormatRequestForUrb и X5WdfRequestSend
Пакет URB (синхронный)	WdfUsbTargetPipeSendUrbSynchronously
На запись (асинхронный)	WdfUsbTargetPipeFormatRequestForWrite и X5WdfRequestSend
На запись (синхронный)	WdfUsbTargetPipeWriteSynchronously

В приводимом в этом разделе примере показано, как драйвер отправляет запрос на чтение каналу получателя USB, после чего получает результаты запроса. Исходный код этого примера адаптирован из кода в файле Osrusbfx2\Sys\Final\Bulkrwr.c.

Когда драйверу Osrusbfx2 приходит запрос на чтение, инфраструктура ставит его в очередь с последовательной диспетчеризацией, созданной драйвером, и вызывает функцию обратного вызова *EvtIoRead* драйвера. Со своей стороны, функция *EvtIoRead* отправляет запрос каналу получателя USB. В листинге 9.27 приведен полный исходный код функции *EvtIoRead*, за исключением некоторых операторов трассировки.

**Листинг 9.27. Отправление асинхронного запроса на чтение каналу получателя USB в драйвере KMDF**

```
VOID OsrFxEvtIoRead(IN WDFQUEUE Queue,
                     IN WDFREQUEST Request,
                     IN size_t Length)
{
    WDFUSBPIPE pipe;
    NTSTATUS status;
    WDFMEMORY reqMemory;
    PDEVICE_CONTEXT pDeviceContext;
    UNREFERENCED_PARAMETER(Queue);
    if (Length > TEST_BOARD_TRANSFER_BUFFER_SIZE) {
        status = STATUS_INVALID_PARAMETER;
        goto Exit;
    }
    pDeviceContext = GetDeviceContext(WdfIoQueueGetDevice(Queue));
    pipe = pDeviceContext->BulkReadPipe;
    status = WdfRequestRetrieveOutputMemory(Request, &reqMemory);
```

```
if (!NT_SUCCESS(status)) {
    goto Exit;
}
status = WdfUsbTargetPipeFormatRequestForRead(pipe,
                                              Request,
                                              reqMemory,
                                              NULL // Offsets
                                              );
if (!NT_SUCCESS(status)) {
    goto Exit;
}
WdfRequestSetCompletionRoutine(Request,
                               EvtRequestReadCompletionRoutine,
                               pipe);
if (WdfRequestSend(Request,
                    WdfUsbTargetPipeGetIoTarget(pipe),
                    WDF_NO_SEND_OPTIONS) == FALSE) {
    status = WdfRequestGetStatus(Request);
    goto Exit;
}
Exit:
if (!NT_SUCCESS(status)) {
    WdfRequestCompleteWithInformation(Request, status, 0);
}
return;
}
```

В листинге 9.27 драйвер проверяет достоверность параметров, чтобы обеспечить, что количество запрошенных байтов не превышает возможности устройства. Если значение параметра `Length` находится в требуемом диапазоне, драйвер получает указатель на область контекста своего устройства, где он сохранил дескриптор объекта канала USB.

Прежде чем отправить запрос ввода/вывода каналу, драйвер должен отформатировать его. Поэтому драйвер извлекает выходной объект памяти из полученного объекта запроса ввода/вывода и вызывает метод `WdfUsbTargetPipeFormatRequestForRead`, передавая ему в параметрах дескриптор объекта канала, дескриптор объекта запроса ввода/вывода, дескриптор объекта памяти и смещение.

Дальше драйвер регистрирует обратный вызов для завершения ввода/вывода, вызывая метод `WdfRequestSetCompletionRoutine`, после чего отправляет запрос каналу, применяя для этого метод `WdfRequestSend`. Для параметра `RequestOptions` драйвер передает значение `NO_SEND_OPTIONS`, указывая, что запрос должен быть отправлен асинхронно и без тайм-аута. Обратите внимание, что драйвер должен вызвать метод `WdfUsbTargetPipeGetIoTarget`, чтобы получить дескриптор объекта получателя ввода/вывода для канала. Инфраструктура создает объект получателя ввода/вывода, который ассоциируется с каждым каналом, но сам объект канала не является получателем ввода/вывода.

В случае неудачного завершения метода `WdfRequestSend`, драйвер вызывает метод `WdfRequestGetStatus`, чтобы определить причину неудачи, после чего завершает запрос, возвращая значение `FALSE`.

По завершению запроса инфраструктура вызывает функцию обратного вызова завершения ввода/вывода. Эта функция извлекает параметры завершения и завершает запрос. Исходный код функции обратного вызова завершения ввода/вывода показан в листинге 9.28.

**Листинг 9.28. Получение параметров завершения запроса USB в драйвере KMDF**

```

VOID EvtRequestReadCompletionRoutine(
    IN WDFREQUEST           Request,
    IN WDFIOTARGET           Target,
    PWDF_REQUEST_COMPLETION_PARAMS CompletionParams,
    IN WDFCONTEXT            Context)
{
    NTSTATUS     status;
    size_t       bytesRead = 0;
    PWDF_USB_REQUEST_COMPLETION_PARAMS usbCompletionParams;
    UNREFERENCED_PARAMETER(Target);
    UNREFERENCED_PARAMETER(Context);

    status = CompletionParams->IoStatus.Status;
    usbCompletionParams = CompletionParams->Parameters.Usb.Completion;
    bytesRead = usbCompletionParams->Parameters.PipeRead.Length;
    WdfRequestCompleteWithInformation(Request, status, bytesRead);
    return;
}

```

При вызове функции обратного вызова завершения ввода/вывода ей передается несколько параметров, наибольший интерес из которых представляет указатель на структуру `WDF_REQUEST_COMPLETION_PARAMS`. Эта структура содержит объединение, которое предоставляет параметры завершения для различных типов запросов ввода/вывода. Для запросов для устройства USB или для получателя канала ввода/вывода для доступа к данным драйвер использует член `Parameters.Usb` этого объединения.

Поле `Parameters.Usb` содержит структуру `WDF_USB_REQUEST_COMPLETION_PARAMS`, которая также является объединением, в которой каждый член описывает результаты для отдельной комбинации типов запросов и типов получателей. Чтобы получить доступ к результатам запроса на чтение, драйвер использует член `Parameters.PipeRead` этого объединения. Поле `Length` этого члена содержит число переданных байтов. Драйвер извлекает это значение и передает его методу `WdiRequestCompleteWithInformation`.

## **Средство непрерывного считывания USB в KMDF**

KMDF предоставляет средство непрерывного считывания (continuous reader), посредством которого драйвер может непрерывно и асинхронно считывать данные из канала USB. Средство непрерывного считывания обеспечивает, что в канале всегда имеется запрос на чтение и, следовательно, что драйвер всегда готов получать данные от устройства.

Драйвер конфигурирует средство непрерывного чтения для входного канала, вставляя специальный код в несколько функций обратного вызова. А именно:

- ◆ функция обратного вызова `EvtDevicePrepareHardware` должна вызывать метод `WdfUsbTargetPipeConfigContinuousReader`. Этот метод ставит в очередь получателя ввода/вывода устройства набор запросов на чтение;
- ◆ чтобы запустить средство непрерывного чтения, функция обратного вызова `EvtDeviceD0Entry` должна вызывать метод `WdfIoTargetStart`;
- ◆ чтобы остановить средство непрерывного чтения, функция обратного вызова `EvtDeviceD0Exit` должна вызывать метод `WdfIoTargetStop`.

При любом наличии данных от устройства получатель ввода/вывода завершает запрос на чтение, и инфраструктура вызывает одну из следующих функций обратного вызова:

- ◆ `EvtUsbTargetPipeReadComplete`, если получатель ввода/вывода успешно считал данные;
- ◆ `EvtUsbTargetPipeReadersFailed`, если чтение данных получателем ввода/вывода завершилось ошибкой.

### Когда запускать средство непрерывного чтения?

Когда я стал членом группы по разработке KMDF, я знал, что необходимо было реализовать средство непрерывного чтения для получателя ввода/вывода канала USB. До того как я присоединился к команде, я разработал и выполнил отладку нескольких специализированных реализаций, и всегда в этой области были проблемы. Проблема с инфраструктурным средством непрерывного чтения, которую мне было необходимо решить, состояла в том, когда начинать чтение. Были три возможные точки.

- В функции обратного вызова `EvtDevicePrepareHardware` при создании средства чтения. Но в данный момент устройство еще не было запитано, поэтому нельзя начинать ввод/вывод в этой точке.
- Непосредственно перед вызовом функции обратного вызова `EvtDeviceD0Entry`. Здесь проблема состояла в том, что драйвер мог бы получать завершения ввода/вывода в то время, когда устройство находилось в процессе входа в рабочее состояние, что было бы чрезмерной нагрузкой для драйвера.
- Сразу же после возвращения управления функцией `EvtDeviceD0Entry`. Но и здесь была проблема — если данные от средства считывания были бы необходимы для реализации включения рабочего режима энергопотребления, то их нельзя было получить.

Рассмотрев все эти сценарии, мы осознали, что важным было не способность инфраструктуры автоматически изменять состояние получателя, а ее способность предоставить разработчикам драйверов возможность запускать и останавливать чтение синхронно<sup>1</sup> по своему собственному усмотрению. Предоставляя разработчикам драйверов компоновочные блоки для создания средства беспрерывного чтения, инфраструктура разрешила многие сценарии, которые были бы невозможны, если бы она предоставляла только автоматическое поведение. (Даун Холэн (*Down Holan*), команда разработчиков *Windows Driver Foundation, Microsoft*.)

## Рекомендация для отправления запросов ввода/вывода

Во всех возможных случаях драйверы должны использовать инфраструктурные интерфейсы для создания, отправления и завершения запросов ввода/вывода, для использования получателей ввода/вывода и для взаимодействия с устройствами USB. Инфраструктурные интерфейсы предоставляют дополнительные возможности для контроля ошибок и проверки достоверности параметров, а также такие дополнительные, особенно полезные для драйверов, возможности, как мониторинг состояния получателя ввода/вывода.

Если драйвер создает запросы ввода/вывода, придерживайтесь следующих правил для доступа к объектам памяти и нижележащим в стеке буферам в этих запросах:

- ◆ не пытайтесь обращаться к буферу, лежащему ниже объекта памяти, после завершения ассоциированного с ним запроса ввода/вывода;

<sup>1</sup> Так как в моих предыдущих проектах остановка считывания была тем моментом, в который существовала тенденция к возникновению состояния гонок.

- ◆ для асинхронных запросов ввода/вывода выделяйте буферы ввода/вывода одновременно с объектами памяти WDF;
- ◆ не завершайте запросы ввода/вывода, созданные драйвером.

Драйверы KMDF могут повторно использовать объекты ввода/вывода. При этом должны соблюдаться следующие правила:

- ◆ если драйвер повторно использует объект запроса WDF, то прежде чем вызывать метод для форматирования объекта под использование в новом запросе ввода/вывода, драйвер должен повторно инициализировать этот объект, вызывая метод `WdfRequestReuse`;
- ◆ если повторно используемый объект запроса WDF содержит объект памяти WDF, который драйвер получил из другого запроса, драйвер должен вызывать метод `WdfRequestReuse` после того, как процесс завершения нового запроса дойдет обратно к драйверу, но перед тем, как драйвер завершит запрос, из которого он получил объект памяти.

# ГЛАВА 10

## Синхронизация

Windows является операционной системой с вытесняющей многозадачностью. Это означает, что несколько потоков могут одновременно пытаться получить доступ к общим данным или ресурсам, и что в одно и то же время (или, иными словами, параллельно) может исполняться несколько драйверных процедур. Чтобы обеспечить целостность данных, драйверы должны синхронизировать доступ к перезаписываемым общим данным. Разработчик драйверов должен определить, какие структуры данных требуют синхронизации доступа и какие методы синхронизации подходят для каждой ситуации.

WDF предоставляет возможности по обеспечению основных требований синхронизации для драйверов. Инфраструктура также предоставляет методы для конфигурирования синхронизации драйвером. Но большинство драйверов также должны использовать базисные элементы синхронизации, предоставляемые Windows. В этой главе дается краткое описание основных требований синхронизации доступа для драйверов и описываются возможности синхронизации, предоставляемые инфраструктурами.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Featured Toaster	%wdk%\src\kmdf\toaster\func\featured
Pcidrv	%wdk%\src\kmdf\Pcidrv
Serial	%wdk%\src\Kmdf\Serial
USB Filter	%wdk%\src\umdf\usb\filter
<b>Документация WDK</b>	
Раздел "Synchronization Techniques" <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80898">http://go.microsoft.com/fwlink/?LinkId=80898</a>
<b>Прочее</b>	
Раздел <b>Locks, Deadlocks, and Synchronization</b> <sup>2</sup> на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=82717">http://go.microsoft.com/fwlink/?LinkId=82717</a>
Раздел "Synchronization" в MSDN	<a href="http://go.microsoft.com/fwlink/?LinkId=80899">http://go.microsoft.com/fwlink/?LinkId=80899</a>

<sup>1</sup> Методы синхронизации. — *Пер.*

<sup>2</sup> Блокировки, взаимоблокировки и синхронизация. — *Пер.*

## Когда требуется применение синхронизации

Синхронизация обеспечивает ситуацию, когда только один поток может иметь доступ к общим данным в любой момент времени и предотвращает вытеснение или прерывание потока драйвера во время критических операций. Синхронизация требуется в следующих обстоятельствах:

- ◆ для любых общих данных, к которым могут обращаться несколько потоков, если только все потоки не обращаются к этим данным только для чтения;
- ◆ для любой операции, состоящей из нескольких действий, которые необходимо выполнить в непрерывной, атомарной последовательности, т. к. другой поток может использовать или изменить данные или ресурсы, требуемые операцией.

В системах с вытесняющей многозадачностью, таких как Windows, один поток может вытесниться другим в любое время. Поэтому, эти требования синхронизации распространяются как на однопроцессорные, так и на многопроцессорные системы.

Каждый драйвер должен быть рассчитан на управление параллельными операциями. Рассмотрим следующие распространенные примеры.

- ◆ **Несколько одновременных запросов ввода/вывода.** Любое устройство — даже устройство, открытое для исключительного доступа — может иметь несколько активных запросов ввода/вывода одновременно. Процесс может выдать перекрывающие друг друга запросы, или несколько потоков могут выдать запросы.
- ◆ **Прерывания, процедуры DPC и другие отложенные вызовы процедур.** Некоторые действия драйверов могут служить причиной асинхронных обратных вызовов функций. Все эти функции могут исполняться параллельно с другими ветвями кода драйвера.

### **Взгляды членов команды разработчиков WDF корпорации Microsoft на синхронизацию**

Люди недооценивают параллельность. Будьте консервативны во время разработки — блокируйте все по умолчанию, особенно в путях, не относящихся к вводу/выводу, где производительность не является критическим вопросом. Оптимизировать пути Plug and Play и энергопотребления — просто глупость. Оптимизацией на производительность можно заняться после того, как вы отладили драйвер и он работает должным образом. (*Нар Ганапати (Nar Ganapathy), команда разработчиков Windows Driver Foundation, Microsoft.*)

Дьявол — в деталях. (*Даун Холэн (Down Holan), команда разработчиков Windows Driver Foundation, Microsoft.*)

Всегда полагайте, что самое худшее обязательно произойдет. (*Питер Вильт (Peter Wielant), команда разработчиков Windows Driver Foundation, Microsoft.*)

## Пример синхронизированного доступа к общим данным

Чтобы понять важность синхронизации, рассмотрим исключительно простую ситуацию, в которой два потока одновременно пытаются увеличить одну и ту же глобальную переменную. Для выполнения этой операции могут потребоваться следующие команды процессора:

1. Считать значение переменной MyVar в регистр.
2. Добавить 1 к значению в регистре.
3. Записать значение в регистре в переменную MyVar.

Если оба потока исполняются одновременно на многопроцессорной системе, на которой не применяется блокировка, операции с взаимоблокировкой или другими средствами синхронизации, состояние гонок может вызвать потерю результатов операции обновления. Например, допустим, что начальное значение переменной `MyVar` равно 0 и что операции выполняются в порядке, показанном на рис. 10.1.

Поток А на процессоре 1...	R1	MyVar	R2	Поток В на процессоре 2...
Считываем переменную <code>MyVar</code> в регистр на процессоре 1	0	0		
	0	0	0	Считываем переменную <code>MyVar</code> в регистр на процессоре 2
	0	0	1	Добавляем 1 к значению регистра процессора 2
	0	1		Записываем значение в регистре процессора 2 в переменную <code>MyVar</code>
Добавляем 1 к значению регистра процессора 1	1	1		
Записываем значение в регистре процессора 1 в переменную <code>MyVar</code>		1		

Рис. 10.1. Исполнение потоков без блокировки на многопроцессорной системе

После того как оба потока увеличат начальное значение переменной `MyVar`, ее значение должно быть равным 2. Но результаты операций потока В на процессоре 1 теряются, когда поток А на процессоре 1 увеличивает первоначальное значение переменной `MyVar`, а потом перезаписывает ее, оставляя конечное значение переменной равным 1. Данная ситуация является примером, когда два потока манипулируют одними и теми же данными в состоянии гонок.

Такая же ситуация с гонками может также возникнуть и на однопроцессорной системе, если поток В вытеснит поток А. Когда система вытесняет поток, она сохраняет значение регистров процессора для этого потока, и восстанавливает их при возобновлении работы потока.

Пример возникновения состояния гонок в результате вытеснения потока показан на рис. 10.2. Как и в предыдущем примере, допустим, что начальное значение переменной `MyVar` равно 0.

Как и в случае с однопроцессорной системой, конечное значение переменной `MyVar` равно 1 вместо 2.

В обоих примерах использование блокировки для синхронизации доступа к переменной решает проблему, вызываемую состоянием гонок. Блокировка обеспечивает, что поток А завершит обновление переменной, прежде чем поток В может получить к ней доступ (см. рис. 10.3).

Блокировка переменной одним потоком обеспечивает, что этот поток сможет завершить свои операции с этой переменной, прежде чем другой поток может получить доступ к ней. С применением блокировки, после исполнения этих двух фрагментов кода значение переменной `MyVar` равно 2, что является правильным и ожидаемым результатом операций.

Хотя упрощен до минимума, этот пример иллюстрирует основную проблему, справиться с которой должен быть способным каждый драйвер. На однопроцессорных системах поток может быть вытеснен или прерван другим потоком, который модифицирует те же самые

данные. На многопроцессорных системах, два или несколько потоков, исполняющихся на разных процессорах, также могут пытаться модифицировать те же самые данные одновременно.

Поток А...	R1	MyVar	R2	Поток В...
Считываем переменную MyVar в регистр	0	0		
Вытесняем поток А и начинаем исполнять поток В				
	0	0	0	Считываем переменную MyVar в регистр
	0	0	1	Добавляем 1 к значению регистра
	0	1		Записываем значение в регистре в переменную MyVar
Вытесняем поток В и начинаем исполнять поток А				
Добавляем 1 к значению регистра	1	1		
Записываем значение в регистре в переменную MyVar		1		

Рис. 10.2. Исполнение потоков без блокировки на однопроцессорной системе

Поток А...	R1	MyVar	R2	Поток В...
Пытаемся получить блокировку		0		
Получили блокировку		0		Пытаемся получить блокировку
Считываем переменную MyVar в регистр	0	0		Ждем
Добавляем 1 к значению регистра	1	0		Ждем
Записываем значение в регистре в переменную MyVar		1		Ждем
Освободили блокировку		1		Получили блокировку
		1	1	Считываем переменную MyVar в регистр
		1	2	Добавляем 1 к значению регистра
		2		Записываем значение в регистре в переменную MyVar
		2		Освободили блокировку

Рис. 10.3. Исполнение потоков с блокировкой на системе с любым количеством процессоров

## Требования синхронизации для драйверов WDF

В отличие от многих приложений, драйверы исполняются нелинейным образом. Драйверные функции допускают повторное выполнение, и драйверам часто приходится обслуживать

одновременно несколько запросов ввода/вывода от нескольких приложений. Вот несколько примеров ситуаций, при которых в драйвере может требоваться применение синхронизации:

- ◆ обеспечение однородных результатов при чтении и записи структур данных, разделяемых несколькими драйверными функциями;
- ◆ обеспечение соответствия ограничениям, накладываемых устройством на число одновременных операций;
- ◆ обеспечение атомарности операций при чтении и записи регистров устройств;
- ◆ контролирование состояний гонок при завершении и отмене запросов ввода/вывода;
- ◆ контролирование состояний гонок при удалении устройства или выгрузке драйвера;
- ◆ запрет повторного выполнения таких операций, как перечисление устройств шины (bus enumeration).

Разные ситуации требуют применения разных методов. Какой способ окажется самым лучшим для применения в конкретной ситуации, будет зависеть от типа данных, к которым обращается драйвер, типа доступа, требуемого драйвером, других компонентов, с которыми он разделяет доступ к данным, и для драйверов режима ядра, уровня IRQL, на котором драйвер обращается к данным. Каждый драйвер имеет свои уникальные требования синхронизации.

### Примечание

При обсуждении управления параллельным доступом к разделяемым данным в этой главе используются два термина: **синхронизация** (synchronization) и **сериализация**<sup>1</sup> (serialization). Термин "синхронизация" применяется в качестве общего термина, обозначающего управление операциями, совместно использующими данные или ресурсы. А термин "сериализация" применяется специфично для обозначения параллельности элементов определенного класса, например, запросов ввода/вывода, и текущего исполнения функций обратного вызова. СерIALIZированные функции обратного вызова не исполняются параллельно. Если две функции сериализированы, то инфраструктура вызовет вторую только после возвращения управления первой.

## Возможности синхронизации, предоставляемые WDF

Инфраструктура WDF разрабатывалась с целью позволить уменьшить в драйверах объем специализированного кода, требуемого для обеспечения синхронизации. Инфраструктуры содержат встроенную функциональность синхронизации посредством следующих возможностей:

- ◆ поддержки счетчиков ссылок и предоставления иерархической объектной модели;
- ◆ автоматической сериализации обратных вызовов функций для событий Plug and Play и энергопотребления;
- ◆ конфигурируемый драйвером поток управления для очередей ввода/вывода;
- ◆ блокировки представления объекта для объектов устройств и очередей ввода/вывода.

<sup>1</sup> Também называется *последовательным упорядочиванием*. — Пер.

Возможности синхронизации, предоставляемые инфраструктурой WDF, предназначены позволить разработчику быстро получить работающий драйвер, с тем, чтобы он мог концентрироваться на оптимизации драйвера для конкретного устройства и для наиболее распространенных сценариев использования. Инфраструктура WDF предоставляет простые возможности синхронизация, не охватывающие всех необходимых требований.

Для многих драйверов требуются примитивы синхронизации, не предоставляемые инфраструктурами, например блокировка чтения/записи. В таких ситуациях драйверы должны использовать механизмы синхронизации, предоставляемые Windows.

В общем, драйверы WDF должны использовать возможности синхронизации, предоставляемые WDF, для объектов и обратных вызовов WDF, а механизмы синхронизации, предоставляемые Windows, для координирования деятельности с приложениями и для выполнения арифметических или логических операций на одной переменной.

В главе 14 объясняется, каким образом выходить за пределы инфраструктуры WDF для вызова функций Windows API и вспомогательные функции драйверов режима ядра.

## Подсчет ссылок и иерархическая объектная модель

Объектная модель WDF устраняет большинство требований синхронизации, связанные с завершением запросов ввода/вывода, и удаления объектов при удалении устройства или выгрузке драйвера. Подсчет ссылок предотвращает удаление объекта на протяжении его использования, а объектная иерархия обеспечивает, что родительские объекты существуют, пока существуют их дочерние объекты. Драйвер не обязан использовать блокировку, чтобы предотвратить удаление объекта, т. к. вместо этого он может просто получить ссылку на объект. Но следует помнить, что состояние объекта может измениться, даже если драйвер удерживает на него ссылку. Например, наличие неосвобожденной ссылки не предотвратит завершение — и освобождение — пакета IRP, на котором базируется объект запроса. Ссылка просто обеспечивает достоверность дескриптора объекта запроса.

Информацию об объектной иерархии и времени жизни объектов см. в главе 5.

## Сериализация обратных вызовов функций для событий Plug and Play и энергопотребления

Обе инфраструктуры автоматически сериализуют большинство функций обратного вызова для событий Plug and Play и энергопотребления, чтобы только в данный момент времени лишь одна такая функция исполнялась для каждого объекта устройства.

Ни UMDF, ни KMDF не сериализуют функции обратного вызова для неожиданного удаления, удаления по запросу и остановки по запросу с другими функциями обратного вызова для событий Plug and Play и энергопотребления, хотя эти функции сериализуются по отношению друг к другу. Поэтому инфраструктура может вызывать эти функции в процессе перехода устройства в другое состояние энергопотребления или когда оно не находится в рабочем состоянии энергопотребления. Далее приведен список функций обратного вызова для неожиданного удаления, удаления по запросу и остановки по запросу:

- ◆ для драйверов UMDF — методы `IPnpCallback::OnSurpriseRemoval`, `OnQueryRemove` и `OnQueryStop` объекта обратного вызова устройства;
- ◆ для драйверов KMDF: `EvtDeviceSurpriseRemoval`, `EvtDeviceQueryRemove` и `EvtDeviceQueryStop`.

Автоматическая сериализация функций обратного вызова для событий Plug and Play и энергопотребления вместе с объектной иерархией WDF означает, что для большинства драйверов требуется незначительный, или совсем никакой, объем кода для обеспечения синхронизации для остановки и удаления объектов.

## Управление потоком для очередей ввода/вывода

Драйверы WDF могут сконфигурировать любую из своих очередей для параллельной, последовательной или ручной диспетчеризации. Анализируя возможности устройства и конфигурируя очереди соответствующим образом, можно уменьшить требования драйвера для дополнительной синхронизации. Метод диспетчеризации очереди ввода/вывода оказывает влияние на степень параллельности в обработке драйвером ввода/вывода, т. к. от него зависит число запросов из очереди, которые могут быть одновременно активными в драйвере.

Рассмотрим примеры.

- ◆ Если устройство способно обрабатывать только один запрос ввода/вывода в данный момент времени, следует сконфигурировать только одну очередь ввода/вывода с последовательной диспетчеризацией.
- ◆ Если устройство может одновременно обрабатывать один запрос на чтение и один запрос на запись, но неограниченное количество запросов IOCTL, то для запросов на чтение и запись можно сконфигурировать по одной очереди с последовательной диспетчеризацией для каждого, а для запросов IOCTL — очередь с параллельной диспетчеризацией.
- ◆ Возможна ситуация, когда устройство способно обрабатывать некоторые запросы IOCTL параллельно, а другие — последовательно. В таком случае можно организовать одну очередь с параллельной диспетчеризацией для входящих запросов IOCTL, проверять их при отправке из этой очереди и отправлять запросы, требующие последовательной обработки, в очередь с последовательной диспетчеризацией для дальнейшей обработки.

Для многих драйверов управление потоком запросов ввода/вывода является наиболее легким и наиболее важным способом для управления параллельными операциями. Но ограничение количества параллельных активных запросов ввода/вывода не разрешает все возможные проблемы синхронизации при обработке ввода/вывода. Например, большинство драйверов требуют дополнительных мероприятий по синхронизации для решения проблемы состояния гонок при отмене ввода/вывода.

Подробная информация о конфигурировании и типах диспетчеризации очередей ввода/вывода представлена в *главе 8*.

## Блокировка представления объекта

Блокировка представления объекта — также называемая блокировкой для синхронизации доступа к объекту — является центральным элементом в схеме синхронизации инфраструктуры WDF. Инфраструктура создает одну блокировку представления объекта для объекта устройства и одну блокировку представления объекта для каждого объекта очереди. Инфраструктура захватывает блокировку устройства или очереди, в зависимости от ситуации, для выполнения сериализации обратных вызовов функций для событий очереди, файла и устройства драйвера.

Инфраструктура захватывает блокировку представления объекта, прежде чем она вызывает большинство функций обратного вызова по событию для объекта устройства или очереди.

Соответственно, функции обратного вызова могут обращаться к объекту и его области контекста без каких-либо дополнительных блокировок. Инфраструктура также использует блокировки представления объекта для реализации области синхронизации, которая описывается в разд. "Область синхронизации и сериализация функций обратных вызовов ввода/вывода" далее в этой главе.

Кроме этого, драйвер может захватить блокировку представления объекта для объекта устройства или очереди, вызывая инфраструктурный метод для захвата блокировки. Эта возможность особенно полезна, когда драйвер обращается к совместно используемым перезаписываемым данным, которые хранятся в объекте устройства или очереди, из кода, находящегося вне сериализованных функций обратного вызова по событию. После захвата блокировки драйвер может безопасно использовать перезаписываемые данные в объекте и выполнять другие действия, оказывающие воздействие на объект. Например, инфраструктура не выполняет сериализации функций обратного вызова завершения ввода/вывода. Соответственно, функция драйвера обратного вызова для завершения ввода/вывода может захватить блокировку представления объекта перед тем, как записывать разделяемые данные в область контекста объекта. Но следует помнить, что драйвер не обязан предохранять каждый доступ к данным в области контекста.

Инфраструктуры предоставляют следующие методы для захвата и освобождения блокировок представления объектов устройства или очереди.

- ◆ Драйверы UMDF вызывают методы `IWDFObject::AcquireLock` и `IWDFObject::ReleaseLock` на инфраструктурных объектах устройства и очереди.

Если драйвер предоставляет недействительный указатель на интерфейс или указатель на интерфейс объекта неправильного типа, методы UMDF вызывают ошибку останова драйвера.

- ◆ Драйверы KMDF вызывают методы `WdfObjectAcquireLock` и `WdfObjectReleaseLock`, передавая им дескриптор объекта типа `WDFDEVICE` или `WDFQUEUE`.

Если драйвер предоставляет недействительный дескриптор или указатель на объект неправильного типа, методы UMDF вызывают останов `bugcheck`.

## Область синхронизации и сериализация функций обратных вызовов ввода/вывода

Для функций обратного вызова, одновременно работающих на одном и том же объекте, может требоваться доступ к специфичным для объекта разделяемым данным или же им может потребоваться пользоваться этими данными совместно со вспомогательными функциями. Например, функции обратного вызова драйвера для очистки и отмены часто используют те же самые данные, как и его функции обратного вызова `ICOTL`. Для управления параллельным исполнением этих функций драйвер может установить область синхронизации.

### Внимание!

Ни UMDF, ни KMDF не сериализуют функции обратного вызова завершения ввода/вывода. Такие функции обратного вызова могут исполняться параллельно с любыми другими функциями обратного вызова драйвера. Если функция обратного вызова завершения ввода/вывода разделяет данные с функцией обратного вызова для другого события ввода/вывода и требует синхронизировать доступ к этим данным, драйвер должен явным образом захватить соответствующие блокировки.

Область синхронизации определяет уровень в объектной иерархии, на котором инфраструктура захватывает блокировку представления объекта. Это, в свою очередь, определяет способ вызова инфраструктурой определенные функции обратного вызова для событий ввода/вывода — параллельно или последовательно. Сериализованные функции обратного вызова для событий ввода/вывода зависят от области синхронизации и инфраструктуры (т. е. от UMDF и KMDF),

В WDF определены следующие три области синхронизации.

◆ **Отсутствие области.**

Отсутствие области синхронизации означает, что WDF не получает блокировку представления объекта и поэтому может вызывать любую функцию обратного вызова по событию ввода/вывода параллельно с любой другой функцией обратного вызова по событию. Отсутствие области синхронизации является установкой по умолчанию для KMDF. Драйвер KMDF может избрать участвовать в сериализации для объекта устройства или очереди, устанавливая область синхронизации на уровне устройства или область синхронизации на уровне очереди при создании им соответствующего объекта.

◆ **Область синхронизации на уровне устройства.**

Область синхронизации на уровне устройства означает, что WDF получает блокировку представления объекта устройства прежде, чем вызывать определенные функции обратного вызова по событию ввода/вывода. Поэтому результатом установки области синхронизации на уровне устройства является последовательно-упорядоченные вызовы определенных функций обратного вызова по событию ввода/вывода для отдельного объекта устройства или для файловых объектов или объектов очереди, являющихся его потомками. Область синхронизации на уровне устройства является настройкой по умолчанию в UMDF.

◆ **Область синхронизации на уровне очереди.**

Область синхронизации на уровне очереди означает, что WDF получает блокировку представления объекта очереди и, таким образом, вызовы функций обратного вызова последовательно упорядочиваются для каждой очереди. Драйвер KMDF может указать область синхронизации на уровне очереди для объекта устройства, с тем, чтобы сериализовать исполнение функций обратного вызова по событию ввода/вывода для каждой очереди, но чтобы функции обратного вызова для разных очередей могли исполняться параллельно.

Первый выпуск UMDF не поддерживает область синхронизации на уровне очереди.

**Внимание!**

UMDF и KMDF имеют разные области синхронизации по умолчанию. Для UMDF областью синхронизации по умолчанию является область синхронизации на уровне устройства. Для KMDF областью синхронизации по умолчанию является отсутствие области.

## **Область синхронизации на уровне устройства и методы диспетчеризации очереди**

Комбинация области синхронизации и метода диспетчеризации очереди предоставляет огромную гибкость в управлении потоком запросов ввода/вывода через драйвер.

На рис. 10.4 показаны объекты, затрагиваемые при установке драйвером области синхронизации на уровне устройства для объекта устройства.



Рис. 10.4. Область синхронизации устройства

На рисунке объект драйвера имеет один объект устройства, а объект устройства — две дочерние очереди и один дочерний файл. Очередь запросов на запись/чтение имеет один дочерний объект запроса. Если драйвер установит на объект устройства область синхронизации на уровне устройства, по умолчанию определенные обратные вызовы объекта устройства, объектов очередей, объекта запроса и файла будут выполняться в последовательном порядке.

Для исследования взаимодействия между областью синхронизации и методом диспетчеризации очереди рассмотрим гипотетическую последовательность ввода/вывода. Допустим, что драйвер на рис. 10.4 сконфигурировал параллельную диспетчеризацию для очереди запросов на запись и чтения и последовательную диспетчеризацию для очереди запросов IOCTL. В табл. 10.1 приводятся действия инфраструктуры по прибытию запросов ввода/вывода для драйвера.

Таблица 10.1. Реакция инфраструктуры на события

Событие	Действие инфраструктуры
Прибывает запрос на чтение 1	Получает блокировку и вызывает функцию обратного вызова чтения ввода/вывода с запросом на чтение 1
Прибывает запрос IOCTL 1	Нет
Прибывает запрос IOCTL 2	Нет
Функция обратного вызова запроса ввода/вывода на чтение возвращает управление	Освобождает блокировку. Получает блокировку и вызывает функцию обратного вызова события IOCTL с запросом IOCTL 1
Прибывает запрос на чтение 2	Нет
Прибывает запрос на запись	Нет

Таблица 10.1 (окончание)

Событие	Действие инфраструктуры
Функция обратного вызова IOCTL отправляет запрос IOCTL 1 получателю ввода/вывода, после чего возвращает управление	Освобождает блокировку. Получает блокировку и вызывает функцию обратного вызова чтения ввода/вывода с запросом на чтение 2
Функция обратного вызова запроса ввода/вывода на чтение возвращает управление	Освобождает блокировку. Получает блокировку и вызывает функцию обратного вызова записи ввода/вывода с запросом на запись

Когда прибывает первый запрос на чтение, инфраструктура получает блокировку представления устройства и отправляет запрос функции обратного вызова для чтения, зарегистрированной очередью. Тем временем прибывают два запроса IOCTL. Инфраструктура добавляет их в очередь запросов IOCTL, но, хотя из этой очереди нет активных запросов, не активирует функцию обратного вызова, т. к. блокировка представления объекта не позволяет других обратных вызовов до тех пор, пока функция обратного вызова запроса чтения не возвратит управление.

Когда функция обратного вызова для запроса чтения возвращает управление, инфраструктура вызывает функцию обратного вызова для запроса IOCTL для обработки первого запроса IOCTL. Функция обратного вызова для запроса IOCTL не может завершить запрос и отправляет его стандартному получателю ввода/вывода, после чего возвращает управление. Но инфраструктура не отправляет следующий запрос ICOTL драйверу, т. к. очередь отправляет запросы последовательно, а драйвер не завершил первый запрос IOCTL и не поставил его в другую очередь.

Пока драйвер обрабатывает запрос IOCTL, прибывает еще один запрос на чтение и один на запись. Инфраструктура отправляет первый из этих запросов функции обратного вызова для события ввода/вывода на очереди для запросов на чтение/запись. Когда эта функция обратного вызова возвращает управление — полагая, что запрос IOCTL в другой очереди еще не был завершен — инфраструктура отправляет драйверу следующий запрос из очереди запросов на чтение/запись. Хотя очередь запросов на чтение/запись и сконфигурирована для параллельной диспетчеризации, блокировка на уровне устройства предотвращает инфраструктуру от одновременного вызова функций обратного вызова для чтения и записи на очереди запросов на чтение/запись.

Хотя применение области синхронизации может значительно упростить драйвер, оно также может вызвать взаимоблокировки. Например, допустим, что в данном примере для завершения запроса IOCTL получателю ввода/вывода требуется дополнительная информация или ему нужно запросить стек устройств драйвера, чтобы выполнить какое-то действие. Если получатель ввода/вывода отправит запрос стеку устройств драйвера в то время, когда первоначальный запрос все еще не завершен, то это вызовет взаимоблокировку драйвера. Инфраструктура не может отправить этот новый запрос драйверу до тех пор, пока не завершится обработка первого запроса.

### Золотое правило синхронизации

При удержании блокировки никогда не делайте вызовов за пределы драйвера.

Основным моментом области синхронизации является то, что ее обычно полезно и безопасно применять в одноуровневых драйверах. Но если драйвер находится в стеке драйверов, выбор области синхронизации становится проблематичным, если только не известно, каким образом ведут себя другие драйверы в стеке и этим поведением можно управлять. (Ильяс Якуб (*Eliyas Yakub*), команда разработчиков *Windows Driver Foundation*, Microsoft.)

## Область синхронизации в драйверах UMDF

Драйверы UMDF могут конфигурировать область синхронизации для функций обратных вызовов своих очередей ввода/вывода и файлах. Для установки области синхронизации драйвер вызывает метод `SetLockingConstraint` интерфейса `IWDFDeviceInitialize` перед тем, как создать объект устройства. Методу передается одно из следующих значений типа `WDF_CALLBACK_CONSTRAINT`, указывающее желаемую область синхронизации:

- ◆ `WdfDeviceLevel` — устанавливает область синхронизации на уровне устройства. Это значение, используемое по умолчанию в UMDF;
- ◆ `None` — область синхронизации не устанавливается.

В табл. 10.2 приведен список интерфейсов обратного вызова и методов UMDF, которые драйвер может реализовать на объекте обратного вызова устройства, очереди и файла, с указанием, получает ли инфраструктура блокировку представления устройства перед тем, как вызвать конкретный метод.

**Таблица 10.2. Функции обратного вызова UMDF с указанием сериализации**

Метод обратного вызова	Сериализация на уровне устройства
<code>IFileCallbackCleanup::OnCleanupFile</code>	Да
<code>IFileCallbackClose::OnCloseFile</code>	Да
<code>IImpersonateCallback::OnImpersonate</code>	Нет
<code>IQueueCallbackCreate::OnCreateFile</code>	Да
<code>IQueueCallbackDefaultIoHandler::OnDefaultIoHandler</code>	Да
<code>IQueueCallbackDeviceIoControl::OnDeviceIoControl</code>	Да
<code>IQueueCallbackIoResume::OnIoResume</code>	Да
<code>IQueueCallbackIoStop::OnIoStop</code>	Да
<code>IQueueCallbackRead::OnRead</code>	Да
<code>IQueueCallbackStateChange::OnStateChange</code>	Да
<code>IQueueCallbackWrite::OnWrite</code>	Да
<code>IRquestCallbackCancel::OnCancel</code>	Да
<code>IRquestCallbackRequestCompletion::OnCompletion</code>	Нет

Обратные вызовы для разных объектов устройств не сериализуются. Таким образом, если драйвер UMDF управляет двумя устройствами и создает объект устройства и одну или несколько очередей ввода/вывода для каждого объекта устройства, инфраструктура может вызывать одновременно функции обратного вызова для этих обоих объектов устройства. Каждый из этих объектов устройства принадлежит отдельному стеку устройств и поэтому находится в отдельном хост-процессе с собственной средой исполнения и, следовательно, без риска возникновения состояния гонок.

При установке драйвером значения области синхронизации `None` UMDF может вызывать любую функцию обратного вызова по событию одновременно с любой другой функцией обратного вызова по событию, и драйвер должен создавать и захватывать блокировки пол-

ностью самостоятельно. Применение таких блокировок в критических секциях программы будет правильным решением, т. к. это предотвращает вытеснение потоков.

Но для многопоточного драйвера вместо блокировки может требоваться применение мьютекса, который можно захватывать рекурсивно.

Образец драйвера USB Filter в файле Device.cpp устанавливает значение области синхронизации равное None следующим образом:

```
FxDeviceInit->SetLockingConstrant (None);
```

Этот драйвер не применяет область синхронизации, т. к. между запросами он не сохраняет никакой информации о состояниях и не требует применения блокировок.

### Внимание!

UMDF не поддерживает никакого метода, с помощью которого драйвер мог бы определить, был ли отменен определенный запрос ввода/вывода. Если драйвер устанавливает блокировку на уровне устройства и обрабатывает все запросы ввода/вывода синхронно, то инфраструктура не может отменить запрос ввода/вывода до тех пор, пока драйвер не освободит блокировку, в результате чего драйвер может перестать отвечать.

Рассмотрим следующий сценарий. Допустим, что драйвер поддерживает запросы IOCTL, состоящие из нескольких команд, и должен сохранять существенный объем информации о состояниях между командами, поэтому функция обратного вызова драйвера OnDeviceControl выполняет операцию синхронно. Но если приложение неожиданно прекращает исполнение, пока запрос остается незаконченным, инфраструктура не может отменить его, т. к. блокировка не будет освобождена до тех пор, пока метод OnDeviceControl не возвратит управление. Таких проблем можно избежать, либо выбирая отсутствие области синхронизации, если драйвер исполняет синхронный ввод/вывод, либо устанавливая тайм-аут для каждого синхронного запроса ввода/ввода.

## Область синхронизации в драйверах KMDF

Драйвер KMDF может сконфигурировать область синхронизации для функций обратного вызова для события ввода/вывода, файла или очереди, устанавливая поле SynchronizationScope в структуре атрибутов объекта при создании им объекта устройства, объекта очереди ввода/вывода или файлового объекта. Инфраструктура может также применять область синхронизации для функций обратных вызовов, активируемых процедурами DPC, таймерами и рабочими элементами, как описано в разд. "Автоматическая сериализация функций обратных вызовов процедурами DPC, таймерами и рабочими элементами" далее в этой главе.

Полю SynchronizationScope можно присвоить следующие допустимые константы структуры WDF\_SYNCHRONIZATION\_SCOPE:

- ◆ WdfSynchronizationScopeDevice — устанавливает область синхронизации на уровне устройства;
- ◆ WdfSynchronizationScopeQueue — устанавливает область синхронизации на уровне очереди;
- ◆ WdfSynchronizationScopeNone — область синхронизации не устанавливается;
- ◆ WdfSynchronizationScopeInheritFromParent — устанавливает для объекта то же самое значение области синхронизации, что и его родительского объекта.

Если драйвер устанавливает область синхронизации для объекта устройства или очереди, то инфраструктура сериализует обратные вызовы функций драйвера для отмены запросов вво-

да/вывода, но не обратные вызовы функций для завершения запросов. Драйвер не может установить область синхронизации для объекта запроса ввода/вывода и должен принять установки по умолчанию для этого поля.

В табл. 10.3 перечислены функции обратного вызова KMDF, для которых применяется область синхронизации. В таблице указывается, какие функции обратного вызова сериализуются при установке драйвером области синхронизации на уровне устройства и какие сериализуются при установке области синхронизации объекта устройства на уровне очереди, а также уровни, на которых каждая функция может исполняться.

Если драйвер принимает область синхронизации по умолчанию и автоматическую сериализацию, то инфраструктура не сериализует никаких функций, приведенных в таблице.

**Таблица 10.3. Сводка последовательного упорядочивания функций обратного вызова KMDF**

Функция обратного вызова	Сериализация на уровне устройства	Сериализация на уровне очереди	Уровень исполнения
CompletionRoutine	Нет	Нет	$\leq$ DISPATCH_LEVEL
EvtDeviceFileCreate	Да	Нет	PASSIVE_LEVEL
EvtDpcFunc	Да*	Да**	DISPATCH_LEVEL
EvtFileCleanup	Да	Нет	PASSIVE_LEVEL
EvtFileClose	Да	Нет	PASSIVE_LEVEL
EvtInterruptDpc	Да*	Нет	DISPATCH_LEVEL
EvtIoCanceledOnQueue	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoDefault	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoDeviceControl	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoInternalDeviceControl	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoQueueState	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoRead	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoResume	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoStop	Да	Да	$\leq$ DISPATCH_LEVEL
EvtIoWrite	Да	Да	$\leq$ DISPATCH_LEVEL
EvtRequestCancel	Да	Да	$\leq$ DISPATCH_LEVEL
EvtTimerFunc	Да*	Да**	DISPATCH_LEVEL
EvtWorkItem	Да*	Да**	PASSIVE_LEVEL

\* Да, если драйвер устанавливает AutomaticSerialization, являющейся настройкой по умолчанию. Автоматическая сериализация рассматривается в разд. "Автоматическая сериализация функций обратных вызовов процедурами DPC, таймерами и рабочими элементами" далее в этой главе.

\*\* Да, если очередь является родителем объекта и если драйвер установит AutomaticSerialization, являющейся настройкой по умолчанию.

## Стандартные настройки области синхронизации

Для объекта драйвера областью синхронизации по умолчанию является `WdfSynchronizationScopeNone` (т. е. отсутствие области синхронизации), а для всех остальных объектов, потомков объекта драйвера — `WdfSynchronizationScopeInheritFromParent` (т. е. наследуемая от родителя). Поэтому инфраструктура по умолчанию не захватывает никаких блокировок и не сериализует никаких обратных вызовов функций, активируемых событиями ввода/вывода, файлами и очередями. Чтобы использовать возможности синхронизации, предоставляемые инфраструктурой, драйвер KMDF должен явно установить область синхронизации на объектах устройства или очереди.

Так как по умолчанию область синхронизации наследуется, драйвер может с легкостью установить область синхронизации для своих очередей ввода/вывода, устанавливая область синхронизации для объекта устройства, который является родителем очередей ввода/вывода.

### Синхронизация на уровне устройства

Если драйвер устанавливает область синхронизации на уровне устройства, то прежде чем активировать обратные вызовы функций для отдельного объекта устройства или для файловых объектов или очередей, являющихся потомками этого объекта устройства, инфраструктура захватывает блокировку представления объекта устройства. Но для обратных вызовов функций для разных объектов устройства сериализация не выполняется, и поэтому данные функции могут исполняться параллельно.

Для использования синхронизации на уровне устройства драйвер присваивает члену `SynchronizationScope` структуры `WDF_OBJECT_ATTRIBUTES` объекта устройства значение `WdfSynchronizationScopeDevice`.

### Синхронизация на уровне очереди

Если драйвер KMDF указывает для объекта устройства область синхронизации на уровне очереди, то инфраструктура исполняет в последовательном порядке функции обратного вызова для событий ввода/вывода, вызываемые объектом очереди, но не функции, вызываемые объектом файла. Соответственно, функции обратного вызова семейства `EvtDeviceFikXxx` и `EvtFikXxx` для объекта устройства и для его очередей могут исполняться параллельно.

Для использования синхронизации на уровне очереди, драйвер присваивает члену `SynchronizationScope` структуры `WDF_OBJECT_ATTRIBUTES` объекта устройства значение `WdfSynchronizationScopeQueue`, а его объекта очереди — `WdfSynchronizationScopeInheritFromParent`. Синхронизация на уровне очереди означает, что только одна из перечисленных в табл. 10.3 функций может исполняться для каждой очереди одновременно. Драйвер не может установить область синхронизации отдельно для каждой очереди.

На рис. 10.5 показаны объекты, затрагиваемые установкой драйвером KMDF синхронизации на уровне очереди для объекта устройства, показанного на рис. 10.4.

Если драйвер устанавливает синхронизацию на уровне очереди, инфраструктура захватывает блокировку представления на уровне очереди. Соответственно, обратные вызовы определенных функций для очереди запросов на запись/чтение и объекта запроса сериализуются по отношению друг к другу. Исполнение функций обратных вызовов для очереди запросов IOCTL сериализуется подобным образом. Но функции обратного вызова для очереди запросов на чтение/запись и для объекта запросов могут исполняться параллельно функциям обратного вызова для очереди запросов IOCTL.



Рис. 10.5. Синхронизация на уровне очереди для драйвера KMDF

Синхронизация на уровне очереди неприменима к файловым объектам. Но по умолчанию файловый объект наследует свою область синхронизации от своего родителя. Поэтому, если драйвер установит синхронизацию на уровне очереди для родительского объекта устройства и зарегистрирует один или несколько обратных вызовов функций для файлового объекта, драйвер должен явным образом установить их область синхронизации или на уровне устройства, или отсутствующей. Если драйвер этого не сделает, инфраструктура генерирует ошибку.

Если драйвер устанавливает область синхронизации для файлового объекта, то он также должен установить уровень исполнения для объекта как `WdfExecutionLevelPassive`, как описано в разд. "Уровни исполнения в драйверах KMDF" далее в этой главе.

Наилучшей практикой для применения с файловыми объектами будет отказ от использования области синхронизации и захват соответствующих блокировок в функциях обратного вызова по событию в ситуациях, требующих синхронизации.

### Пример KMDF: область синхронизации

В листинге 10.1 приведен пример установки образцом драйвера Serial (который управляет последовательными портами) области синхронизации уровня `WdfSynchronizationScopeNone` для своих файловых объектов. Этот исходный код находится в заголовочном файле `Serial\Pnp.c`.

#### Листинг 10.1. Установка области синхронизации в драйвере KMDF

```

WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.SynchronizationScope = WdfSynchronizationScopeNone;
WDF_FILEOBJECT_CONFIG_INIT(&fileobjectConfig,
                           SerialEvtDeviceFileCreate,
                           SerialEvtFileClose,
                           WDF_NO_EVENT_CALLBACK // Cleanup
                           );
  
```

```
WdfDeviceInitSetFileObjectConfig(DeviceInit,
    &fileobjectConfig,
    &attributes);
```

Код в листинге 10.1 размещен в функции обратного вызова `EvtDriverDeviceAdd` драйвера до создания драйвером объекта устройства. Для установки области синхронизации для файловых объектов драйвер инициализирует структуру атрибутов объекта и присваивает ее полю `SynchronizationScope` значение `WdfSynchronizationScopeNone`. Дальше драйвер инициализирует структуру `WDF_FILEOBJECT_CONFIG` указателями на функции обратного вызова файлового объекта драйвера. Наконец, драйвер вызывает метод `WdfDeviceInitSetFileObjectConfig`, чтобы записать конфигурационную информацию файлового объекта и атрибуты объекта в структуре `WDFDEVICE_INIT`, которую он в дальнейшем передаст объекту устройства.

В образце драйвера `Serial` функции обратного вызова `SerialEvtDeviceFileCreate` и `SerialEvtFileClose` для файлового объекта не пользуются данными совместно с другими функциями обратного вызова для объекта устройства, и поэтому им не требуются блокировки. Слово "Serial" в именах функций указывает, что данные функции являются частью об разца драйвера `Serial`, а не то, что они исполняются последовательным образом.

## Автоматическая сериализация функций обратных вызовов процедурами DPC, таймерами и рабочими элементами

Каждый объект процедуры DPC, таймера или рабочего элемента является потомком объекта устройства или объекта очереди. Чтобы упростить реализацию драйверных функций обратного вызова для процедур DPC, таймеров и рабочих элементов, KMDF по умолчанию автоматически выполняет эти функции и функции обратного вызова для родительского объекта последовательно. Драйвер может переопределить настройки по умолчанию, присвоив полю `AutomaticSerialization` значение `FALSE` при создании объекта процедуры DPC, таймера или рабочего элемента.

При разрешенной автоматической сериализации, инфраструктура применяет настройки `SynchronizationScope` объекта устройства или очереди, являющегося родителем данного объекта. Таким образом, если родителем объекта таймера является объект устройства, для которого драйвер установил область синхронизации, то инфраструктура захватывает блокировку объекта устройства, прежде чем делать обратный вызов функции `EvtTimerFunc`.

Автоматическую сериализацию следует применять только для обратных вызовов функций, которые могут исполняться на одинаковом уровне IRQL. Например, если объект устройства имеет дочерний объект процедуры DPC и объект рабочего элемента, то нельзя разрешать автоматическую сериализацию для обоих дочерних объектов, т. к. обратный вызов процедуры DPC всегда исполняется на уровне `IRQL = DISPATCH_LEVEL`, а обратный вызов рабочего элемента всегда исполняется на уровне `IRQL = PASSIVE_LEVEL`. Инфраструктура не может сериализовать обе эти функции, применяя одну и ту же блокировку. Дополнительная информация о влиянии уровня IRQL на сериализацию приводится в разд. "Уровни исполнения в драйверах KMDF" далее в этой главе.

Образец драйвера `Sample` создает процедуры DPC, которые исполняются по истечению их таймеров и когда он завершает операцию чтения или записи. Драйвер разрешает автоматическую сериализацию для всех этих процедур DPC, чтобы они исполнялись последовательно с другими обратными вызовами, предоставляемыми объектом очереди или объектом устройства. В результате процедуры DPC могут обращаться к устройству и областям контекста очередей, не применяя блокировок. В листинге 10.2 приводится пример автоматиче-

ской установки сериализации драйвером Serial для одной из этих процедур DPC. Этот исходный код находится в заголовочном файле KmdfSerialUtils.c.

#### Листинг 10.2. Установка автоматической сериализации для объекта процедуры DPC

```
WDF_DPC_CONFIG_INIT(&dpcConfig, SerialCompleteWrite);
dpcConfig.AutomaticSerialization = TRUE;
WF_OBJECT_ATTRIBUTES_INIT(&dpcAttributes);
dpcAttributes.ParentObject = pDevExt->WdfDevice;
status = WdfDpcCreate(&dpcConfig,
                      &dpcAttributes,
                      &pDevExt->CompleteWriteDpc);
```

В представленном листинге образец драйвера Serial инициализирует структуру `WDF_DPC_CONFIG` именем процедуры DPC, после чего присваивает полю `AutomaticSerialization` этой структуры значение `TRUE`.

Данная процедура DPC используется только с определенным объектом устройства, поэтому драйвер инициализирует структуру атрибутов объекта и устанавливает объект устройства в качестве родителя. После этого драйвер вызывает метод `WdfDpcCreate` для создания процедуры DPC, передавая ему в качестве параметров указатель на структуру конфигурации DPC, структуру атрибутов и процедуру DPC.

### Уровни исполнения в драйверах KMDF

Каждый примитив для реализации блокировок и синхронизации имеет соответствующий уровень IRQL. В некоторых ситуациях для сериализации обратных вызовов инфраструктура применяет синхронизационный примитив, имеющий уровень IRQL = `PASSIVE_LEVEL`, а в других использует примитив, исполняющийся на уровне IRQL = `DISPATCH_LEVEL` или, в случае с прерываниями, на уровне `DIRQL`. Для драйверов KMDF инфраструктуру можно заставить применять блокировку на уровне IRQL = `PASSIVE_LEVEL` при сериализации определенных обратных вызовов.

Устанавливая уровень исполнения, драйверы KMDF могут указать максимальный уровень IRQL, на котором активируются обратные вызовы для объектов драйвера, объектов устройства и для файловых и общих объектов. Подобно области синхронизации, уровень исполнения является атрибутом объекта. Драйвер устанавливает уровень исполнения, присваивая соответствующее значение полю `ExecutionLevel` структуры атрибутов при создании им объекта драйвера, устройства, файла или общего объекта.

KMDF поддерживает следующие уровни исполнения.

- ◆ `WdfExecutionLevelPassive`

Уровень исполнения `passive` (пассивный) означает, что инфраструктура вызывает все функции обратного вызова по событию для объекта на уровне IRQL = `PASSIVE_LEVEL`. При необходимости KMDF активирует обратный вызов из системного рабочего потока.

Драйверы могут установить этот уровень только для объектов устройств и файловых объектов. Обычно драйвер должен установить уровень исполнения `passive`, только если обратные вызовы обращаются к страничному коду или данным или вызывают другие функции, которые должны вызываться на уровне IRQL = `PASSIVE_LEVEL`.

Функции обратного вызова для событий, происходящих на файловых объектах, всегда вызываются на уровне IRQL = `PASSIVE_LEVEL`, т. к. этим функциям необходимо иметь

возможность доступа к страничному коду или данным. Функции обратного вызова *EvtDeviceFileCreate* и *EvtFileCleanup* исполняются в контексте потока приложения, открывшего дескриптор файла. А функцию *EvtFileClose* можно вызывать в контексте произвольного потока.

◆ **WdfExecutionLevelDispatch**

Уровень исполнения *dispatch* означает, что инфраструктура может вызывать функции обратного вызова с любого уровня IRQL до уровня DISPATCH\_LEVEL включительно.

Эта установка не заставляет все обратные вызовы выполняться на уровне DISPATCH\_LEVEL. Но если для исполнения обратного вызова требуется синхронизация, то KMDF использует спин-блокировку, что поднимает уровень IRQL до DISPATCH\_LEVEL. Поэтому такие обратные вызовы должны создаваться для исполнения на уровне DISPATCH\_LEVEL, а для выполнения задач, требующих исполнения на уровне PASSIVE\_LEVEL, следует применять рабочие элементы. Если рабочий элемент пользуется данными совместно с другими функциями обратного вызова, он может применить блокировку представления устройства или очереди для синхронизации доступа. Для получения этой блокировки драйвер вызывает метод *WdfObjectAcquireLock*.

◆ **WdfExecutionLevelInheritFromParent**

Уровень исполнения *inherited* (унаследованный) означает, что инфраструктура применяет уровень исполнения родительского объекта.

Эта установка является стандартной для всех объектов за исключением объекта драйвера. Для объекта драйвера уровнем исполнения по умолчанию является *WdfExecutionLevelDispatch*.

Уровень исполнения определяет механизм, применяемый инфраструктурой в блокировке для представления объекта.

◆ Если драйвер установит уровень исполнения *passive*, то инфраструктура будет сериализовать обратные вызовы функций для событий, используя быстрый мьютекс (fast mutex), что является блокировкой PASSIVE\_LEVEL.

Поэтому инфраструктура вызывает сериализованные функции обратного вызова на уровне PASSIVE\_LEVEL, независимо от области синхронизации.

◆ Если драйвер установит уровень исполнения *dispatch*, то инфраструктура будет сериализовать обратные вызовы функций для событий, используя спин-блокировку (spin lock), что поднимает уровень до DISPATCH\_LEVEL.

Если область синхронизации установлена отсутствующей, то инфраструктура не получает блокировку представления и может активировать функции обратного вызова как на уровне DISPATCH\_LEVEL, так и на уровне PASSIVE\_LEVEL.

Функции обратного вызова *EvtDeviceFileCreate*, *EvtFileCleanup* и *EvtFileClose* исполняются в контексте потока вызывающего клиента и используют страничные данные, поэтому их необходимо вызывать на уровне PASSIVE\_LEVEL. Если драйвер устанавливает область синхронизации для своих файловых объектов, то он должен установить для них уровень исполнения *WdfExecutionLevelPassive*, чтобы заставить инфраструктуру применять блокировку PASSIVE\_LEVEL.

В следующем примере, взятом из файла *Toaster\Func\Featured\Toaster.c*, показано, как образец драйвера *Featured Toaster* устанавливает область синхронизации и уровень исполнения:

```
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&fdoAttributes, FDO_DATA);
```

```
fdoAttributes.SynchronizationScope = WdfSynchronizationScopeDevice;
fdoAttributes.ExecutionLevel = WdfExecutionLevelPassive;
. . . // Код опущен, чтобы сохранить место.
status = WdfDeviceCreateC&DeviceInit, &fdoAttributes, &device);
```

Как можно видеть из предыдущего примера, драйвер устанавливает уровень исполнения в структуре атрибутов, присваивая значение полю ExecutionLevel. Для объекта устройства драйвер устанавливает синхронизацию на уровне устройства и уровень исполнения PASSIVE\_LEVEL, чтобы инфраструктура выполняла сериализацию обратных вызовов функций для событий ввода/вывода и для файлового объекта и вызывала эти функции на уровне IRQL = PASSIVE\_LEVEL.

Хотя на первых порах может казаться, что принудительное исполнение всех обратных вызовов на уровне PASSIVE\_LEVEL может существенно упростить программирование драйверов, для многих драйверов эту установку нельзя применять. В драйвере, выполняющем большой объем работы, обработка ввода/вывода может быть задержана по причине, что обратные вызовы драйвера не смогут исполниться до тех пор, пока не будет завершена вся другая обработка на уровне DISPATCH\_LEVEL.

## Спин-блокировки и wait-блокировки KMDF

В KMDF определены объекты блокировок двух типов: wait-блокировки (`WDFWAITLOCK`) и спин-блокировки (`WDFSPINLOCK`). Wait-блокировки применяются для синхронизации операций на уровне PASSIVE\_LEVEL, а спин-блокировки — на уровне DISPATCH\_LEVEL или выше. Для блокировок обоих типов инфраструктура реализует обнаружение взаимоблокировок и отслеживает историю захвата блокировок.

### Wait-блокировки

Wait-блокировка представляет собой механизм синхронизации уровня PASSIVE\_LEVEL, похожий на объект события Windows. Ее нельзя использовать рекурсивно; в ситуациях, требующих рекурсивного захвата блокировки уровня PASSIVE\_LEVEL, драйвер должен использовать мьютексы Windows.

Когда драйвер пытается получить wait-блокировку, он может предоставить период тайм-аута, чтобы ограничить время, в течение которого драйвер ожидает блокировку. Если драйвер не предоставляет значение периода тайм-аута, то метод получения блокировки возвращается немедленно. Если драйвер предоставляет значение периода тайм-аута, то он должен получить блокировку на уровне PASSIVE\_LEVEL. Если драйвер предоставляет нулевое значение периода тайм-аута, он может получить блокировку на уровне  $IRQL \leq DISPATCH\_LEVEL$ , т. к. поток не входит в состояние ожидания.

Как правило, драйвер должен установить родителем wait-блокировки объект, предохраняемый блокировкой. Когда предохраняемый объект удаляется, инфраструктура также удаляет блокировку.

Образец драйвера Serial создает область контекста для своего объекта прерывания и считывает и записывает данные в область контекста в функциях обратного вызова `EvtDeviceD0ExitPreInterruptsDisabled` и `EvtDeviceD0EntryPostInterruptsEnabled`. Инфраструктура вызывает эти функции на уровне PASSIVE\_LEVEL, поэтому драйвер создает wait-блокировку для синхронизации доступа к области контекста, как показано в листинге 10.3.

Код примера, приведенного в данном листинге, является частью кода драйвера для обработки функции *EvtDriverDeviceAdd* и находится в файле *Serial\Pnp.c*.

#### Листинг 10.3. Создания wait-блокировки в драйвере KMDF

```
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.ParentObject = pDevExt->WdfInterrupt;
interruptContext = SerialGetInterruptContext(pDevExt->WdfInterrupt);
status = WdfWaitLockCreate(&attributes,
                           &interruptContext->InterruptStateLock);
```

Прежде чем создать объект блокировки, драйвер инициализирует структуру атрибутов объекта и присваивает значению поля *ParentObject* объект прерывания. После этого драйвер извлекает указатель на область контекста объекта прерывания, чтобы сохранить дескриптор объекта блокировки в области контекста. Наконец, драйвер создает объект блокировки, вызывая метод *WdfWaitLockCreate*, передавая ему в параметрах указатель на структуру атрибутов объекта и адрес в области контекста, где сохранить дескриптор блокировки.

В листинге 10.4 приведен пример использования драйвером блокировки для защиты от доступа к области контекста. Этот код взят из функции обратного вызова *EvtDeviceD0EntryPostInterruptsEnabled* в файле *Serial\Pnp.c*.

#### Листинг 10.4. Использование wait-блокировки в драйвере KMDF

```
WdfWaitLockAcquire(interruptContext->InterruptStateLock, NULL);
interruptContext->IsInterruptConnected = TRUE;
WdfWaitLockRelease(interruptContext->InterruptStateLock);
```

В представленном листинге драйвер захватывает блокировку без периода тайм-аута, таким образом указывая, что он будет ожидать бесконечно. Захватив блокировку, драйвер устанавливает значение *IsInterruptConnected* в области контекста объекта прерывания, после чего освобождает блокировку.

Хотя область контекста ассоциирована с объектом прерывания драйвера, драйвер не требует спин-блокировку прерывания для защиты области контекста. В данном примере драйвер считывает и записывает в область контекста только в функциях обратного вызова, которые исполняются на уровне *PASSIVE\_LEVEL*; он никогда не обращается к области контекста на высших уровнях *IRQL*. Инфраструктура сериализует обратные вызовы функции *EvtDeviceD0EntryPostInterruptsEnabled* для объекта устройства, и драйверу не требуется получать рекурсивную блокировку. Поэтому драйвер может использовать wait-блокировку KMDF.

## Спин-блокировки

Слово *spin* по-английски означает "вращаться". Спин-блокировка делает точно то, что подразумевает ее имя: пока один поток владеет блокировкой, все другие потоки, ожидающие получить блокировку, врачаются в цикле до тех пор, пока блокировка не станет доступной. На практике поток захватывает блокировку, записывая в определенную ячейку памяти заранее согласованное значение. Поток владеет блокировкой до тех пор, пока ячейка содержит значение "захвачено". Другие потоки, пытающиеся захватить блокировку, периодически

проверяют содержимое ячейки. Как только содержимое ячейки принимает значение "свободно", блокировка может быть захвачена другим потоком, который устанавливает значение ячейки равным "захвачено". Потоки не блокируются, т. е. они не приостанавливаются или вытесняются. Вместо этого, в многопроцессорных системах каждый поток сохраняет управление центральным процессором, таким образом предотвращая исполнение другого кода на таком же или более низком уровне IRQL.

Сpin-блокировки являются единственным механизмом синхронизации, который можно использовать на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ . Код, удерживающий spin-блокировку, исполняется на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , и это означает, что системный код, выполняющий переключение потоков, не может исполняться, и поэтому текущий поток нельзя вытеснить. Драйверы должны удерживать spin-блокировки только в течение минимально требуемого периода времени и не содержать в пути заблокированного кода никаких задач, не требующих блокировки. Удерживание spin-блокировки в течение длительного времени может ухудшить производительность на уровне системы.

## Виды спин-блокировок

В KMDF определены два вида спин-блокировок:

- ◆ обычные спин-блокировки, работающие на уровне `DISPATCH_LEVEL`;
- ◆ спин-блокировка прерывания, которая сериализует два обратных вызова функции `EvtInterruptIsr` на уровне `DIRQL`.

Если драйвер создает два или больше объекта прерываний и их соответствующие функции обратного вызова `EvtInterruptIsr` обращаются к общим данным, необходимо создать спин-блокировку прерывания, чтобы сериализовать доступ к данным.

Сpin-блокировки KMDF могут иметь области контекста объекта для хранения драйвером специфичных для блокировки данных. Весь код в спин-блокировке должен соответствовать требованиям для исполнения на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ .

## Пример KMDF: спин-блокировки

Образец драйвера Pcidrv создает три спин-блокировки. Эти спин-блокировки защищают определенные поля области контекста объекта устройства, список буферов для отправления данных и список буферов для получения данных. В листинге 10.5 показан пример создания драйвером Pcidrv спин-блокировки для защиты объекта устройства. Исходный код этого примера взят из файла Pcidrv\Sys\Hw\Nic\_init.c.

### Листинг 10.5. Создания спин-блокировки в драйвере KMDF

```
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.ParentObject = FdoData->WdfDevice;
status = WdfSpinLockCreate(&attributes, &FdoData->Lock);
if (!NT_SUCCESS(status)) {
    return status;
}
```

Чтобы создать объект блокировки, для которого родителем является объект устройства, драйвер инициализирует структуру атрибутов объекта и присваивает значению поля `ParentObject` дескриптор объекта устройства. После этого драйвер вызывает метод

WdfSpinLockCreate, передавая ему в параметрах структуру атрибутов и указатель на переменную для получения дескриптора объекта спин-блокировки. Драйвер сохраняет дескриптор объекта спин-блокировки в поле Lock области контекста объекта устройства, на которую указывает FdoData.

Образец драйвера Pcidrv не конфигурирует область синхронизации. Вместо этого, он принимает стандартную настройку отсутствия синхронизации и использует блокировки для защиты структур данных от одновременного доступа. В листинге 10.6 показан пример захвата драйвером блокировок, которые защищают объект устройства и буфер для получения данных, прежде чем обращаться к ним. Этот исходный код находится в заголовочном файле Pcidrv\Sys\Hw\Nic\_req.c.

#### Листинг 10.6. Захват и освобождение спин-блокировок

```
WdfSpinLockAcquire(FdoData->Lock);
WdfSpinLockAcquire(FdoData->RcvLock);
if (MP_TEST_FLAC(FdoData, fMP_ADAPTER_LINK_DETECTION)) {
    status = WdfRequestForwardToIoQueue(Request,
                                         FdoData->PendingIoctlQueue);
    WdfSpinLockRelease(FdoData->RcvLock);
    WdfSpinLockRelease(FdoData->Lock);
    . . . // Код опущен, чтобы сохранить место.
}
WdfSpinLockRelease(FdoData->RcvLock);
WdfSpinLockRelease(FdoData->Lock);
```

В представленном листинге блокировка, которая сохраняется по FdoData->Lock, защищает объект устройства, а блокировка, которая сохраняется по FdoData->RcvLock, защищает список буферов для получения данных. Драйвер захватывает обе блокировки. Для захвата блокировки драйвер вызывает метод WdfSpinLockAcquire, передавая ему дескриптор объекта блокировки. Для освобождения блокировки драйвер вызывает метод WdfSpinLockRelease, также передавая ему дескриптор объекта блокировки. Во избежание возможных взаимоблокировок, драйвер освобождает блокировки в порядке, обратном их захвату.

#### Совет

Если драйвер использует несколько блокировок, необходимо всегда захватывать их в одном и том же порядке, а освобождать в порядке, обратном порядку захвата. То есть, если драйвер захватывает блокировку А перед блокировкой В, он должен освободить блокировку В перед блокировкой А. В противном случае существует возможность возникновения взаимоблокировки. Также при работе с инструментом Driver Verifier (DV) необходимо включить функцию Deadlock Detection. Таким образом, DV сможет выполнить проверку на возможные нарушения иерархии блокировок.

Если в предыдущем примере вы обратили внимание на то, что драйвер Pcidrv сохраняет списки буферов для приема и отправки данных в области контекста объекта устройства, вам может показаться странным, что он создает три отдельных блокировок, когда одной блокировки для всего устройства было бы достаточно. Причина этому очень проста — производительность. Предохраняя списки по отдельности, драйвер может избежать захвата более общей блокировки, которой является блокировка объекта устройства, когда ему требуется доступ только к одному списку. Например, при обработке операции приема драйвер захватывает блокировку только для списка буферов получаемых данных, так что другой обрат-

ный вызов драйвера, обрабатывающий операции отправления, может параллельно захватить блокировку для списка буферов отправляемых данных.

## Синхронизация отмены запросов ввода/вывода в драйверах KMDF

Отмена запросов ввода/вывода по своей природе является асинхронной и требует тщательной синхронизации. Хотя для отмены запросов ввода/вывода некоторые драйверы могут полагаться на инфраструктуру, для большинства драйверов требуется реализовывать код для выполнения отмены, по крайней мере, некоторых своих запросов ввода/вывода.

Если драйвер всегда содержит свои длительные запросы ввода/вывода в очереди, а потом быстро их завершает, применения какого-либо кода синхронизации можно избежать. Инфраструктура просто отменяет запросы, пока они находятся в очереди. А после доставки запроса из очереди, драйвер может оставить его в неотменяемом состоянии, т. к. он завершается немедленно по доставке.

Но большинство драйверов удерживают извлеченные из очереди запросы в транзите, пока аппаратура выполняет операцию, и они должны поддерживать отмену запросов.

Синхронизация является наиболее подверженным ошибкам аспектом отмены ввода/вывода в драйверах WDM, т. к. драйвер должен отслеживать, кто владеет запросом в любой момент времени, и, соответственно, является ответственным за его отмену. Хотя отмена запросов является таким же сложным мероприятием и в драйверах WDF, объектная модель инфраструктуры предоставляет два метода, упрощающих отслеживание запросов:

- ◆ синхронизация отмены через область синхронизации;
- ◆ синхронизация отмены с отслеживанием состояния в области контекста запроса.

Кроме этого, если драйвер создает подзапросы для сбора информации, требуемой для завершения доставленного инфраструктурой запроса, то удаление родительского запроса необходимо синхронизировать с удалением его дочерних подзапросов. Объект отмены KMDF предоставляет относительно простой способ отмены подзапросов при отмене родительского запроса.

Все эти способы синхронизации описываются в следующих разделах.

### Синхронизация отмены через область синхронизации

Драйвер обычно завершает длительные запросы ввода/вывода в асинхронных обратных вызовах, таких как, например, отложенный вызов процедуры прерывания, отложенный вызов процедуры таймера, или рабочий элемент. Простым подходом к синхронизации отмены ввода/вывода будет использование синхронизации на уровне устройства или очереди на очереди, а также на асинхронном объекте обратного вызова, завершающего запрос.

Этот подход демонстрируется в образце драйвера WDK под названием Echo. Для запросов на чтение и запись этот драйвер создает очередь с последовательной диспетчеризацией. Для объекта устройства драйвер устанавливает синхронизацию на уровне устройства, а очередь наследует эту область синхронизации по умолчанию. Драйвер также устанавливает очередь родителем объекта таймера, так что отложенный вызов процедуры таймера по умолчанию сериализуется с обратными вызовами функций очереди для событий ввода/вывода. При каждой доставке очередью запроса на чтение или запись драйвер помечает запрос как отме-

няемый и сохраняет его дескриптор в области контекста очереди. Запрос завершается позже отложенным вызовом процедуры таймера.

Так как применяется область синхронизации, то инфраструктура обеспечивает, что процедура отмены и отложенный вызов процедуры таймера не исполняются одновременно. Поэтому процедура отложенного вызова таймера не захватывает блокировку для проверки состояния отмены запроса или для его завершения (листинг 10.7).

#### Листинг 10.7. Отмена запроса с применением синхронизации, предоставляемой инфраструктурой

```
if (queueContext->CurrentRequest!= NULL) {  
    Status = WdfRequestUnmarkCancelable(Request);  
    if ( Status != STATUS_CANCELLED ) {  
        queueContext->CurrentRequest = NULL;  
        Status = queueContext->CurrentStatus;  
        WdfRequestComplete(Request, Status);  
    }  
    else {  
        . . . // Код опущен для краткости.  
    }  
}
```

Драйвер сохраняет дескриптор текущего запроса в области контекста очереди. Значение дескриптора, равное `NULL`, говорит о том, что запрос уже был завершен, поэтому драйвер не должен пытаться отменить его. Если дескриптор достоверный, то драйвер помечает запрос как неотменяемый, вызывая для этого метод `WdfRequestUnmarkCancelable`. Если запрос уже был отменен, то метод `WdfRequestUnmarkCancelable` возвращает `STATUS_CANCELLED`, которое означает, что обратный вызов драйвера для отмены ввода/вывода будет исполняться после возвращения управления отложенным вызовом процедуры таймера. В таком случае драйвер не завершает запрос, т. к. это будет сделано обратным вызовом для отмены запроса. Если запрос еще не был завершен или отменен, то драйвер обновляет информацию в области контекста очереди и вызывает метод `WdfRequestComplete`, чтобы завершить запрос.

## Синхронизация отмены с отслеживанием состояния в области контекста

При другом подходе к отмене запросов ввода/вывода применяется своя собственная блокировка и переменная состояния для отслеживания текущего владельца запроса. Если драйвер использует две последовательные очереди или подобный механизм, при котором только один или два запроса ожидают исполнения в любой момент времени, то можно отслеживать текущего владельца с помощью переменной состояния для каждого незаконченного запроса в области контекста объекта устройства или очереди (см. листинг 10.7).

Но проблема усложняется, если драйвер использует параллельную очередь, и поэтому может иметь несколько запросов в отменяемом состоянии. В таком случае текущий владелец отслеживается с помощью области контекста объекта запроса и подсчета ссылок.

Давайте рассмотрим сценарий, в котором запрос помечается как отменяемый и запускается операция аппаратного ввода/вывода. Аппаратура обычно извещает драйвер о завершении операции, генерируя прерывание. После этого драйвер ставит в очередь процедуру DPC для

завершения запроса. Допустим, что драйвер может прекратить операцию аппаратного ввода/вывода, когда отменяется соответствующий запрос.

Псевдокод реализации синхронизации для такого драйвера приводится в этой последовательности шагов:

1. Создаем область контекста в созданном инфраструктурой запросе, объявляя тип контекста в заголовочном файле следующим образом:

```
typedef struct _REQUEST_CONTEXT {
    BOOLEAN IsCancelled;
    BOOLEAN IsTerminateFailed;
    KSPIN_LOCK Lock;
} REQUEST_CONTEXT, *PREQUEST_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(REQUEST_CONTEXT,
                                    CetRequestContext);
```

Флаг `IsCancelled` в области контекста запроса указывает, был ли запрос отменен, а флаг `IsTerminateFailed` указывает, возвратила ли внутренняя функция драйвера `TerminateIo` статус неудачи.

Для синхронизации завершения запроса применяется спин-блокировка. Выбор блокировки является критическим для производительности драйвера, т. к. блокировка захватывается и освобождается в ветви завершения запроса, как показано дальше в шаге 5, а также в ветви отмены запроса, показанной в шаге 4. Применение `KSPIN_LOCK` вместо `WDFSPINLOCK` позволяет избежать возможных неудачных попыток выделения объекта блокировки WDF в пути ввода/вывода.

2. В функции `EvtDriverDeviceAdd` конфигурируем область контекста для каждого запроса, представляемого инфраструктурой драйверу:

```
EvtDriverDeviceAdd (Driver, DeviceInit)
{
    WDF_OBJECT_ATTRIBUTES attributes;
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE
        (&attributes, REQUEST_CONTEXT);
    WdfDeviceInitSetRequestAttributes(DeviceInit, &attributes);
}
```

3. В функции `EvtIoXXX` получаем дополнительную ссылку на объект запроса и помечаем запрос отменяемым:

```
EvtIoDispatch(Queue, Request)
{
    PREQUEST_CONTEXT reqContext;
    reqContext = CetRequestContext(Request);

    reqContext->IsCancelled = FALSE;
    reqContext->IsTerminateFailed = FALSE;
    KeInitializeSpinLock(&reqContext->Lock);
    WdfObjectReference(Request);
    WdfRequestMarkCancelable(Request, EvtRequestCancelRoutine);
    // Начинаем аппаратную операцию ввода/вывода.
    . . .
}
```

Инфраструктура отправляет запросы из очереди в функцию обратного вызова `EvtIoXXX`. Эта функция инициализирует флаги `IsCancelled` и `IsTerminateFailed` значением `FALSE`. Инициализация флагов не является совершенно необходимой, т. к. инфраструктура инициализирует область контекста нулевыми значениями, но в псевдокоде этот шаг показывается для большей ясности. Эта функция обратного вызова также инициализирует спин-блокировку и получает дополнительную ссылку на объект запроса. Потом она помечает запрос как отменяемый и начинает длительную операцию ввода/вывода.

По завершению операции ввода/вывода устройство генерирует прерывание и функция обратного вызова впоследствии завершает запрос. Дополнительная ссылка позволяет обращаться к области контекста запроса, чтобы узнать состояние запроса даже после его завершения.

4. При отмене запроса инфраструктура вызывает функцию обратного вызова драйвера `EvtRequestCancel`. Для синхронизации завершения запроса в этой функции применяется спин-блокировка:

```
EvtRequestCancelRoutine(Request)
{
    PREQUEST_CONTEXT reqContext = GetRequestContext (Request);
    BOOLEAN completeRequest;
    KIRQL oldIrql;

    KeAcquireSpinlock(&reqContext->Lock, &oldIrql);
    reqContext->IsCancelled = TRUE;
    if (TerminateIO() == TRUE) {
        WdfObjectDereference(Request);
        completeRequest = TRUE;
    }
    else {
        reqContext->IsTerminateFailed = TRUE;
        completeRequest = FALSE;
    }
    KeReleaseSpinlock(&reqContext->Lock, oldIrql);
    if (completeRequest) {
        WdfRequestComplete(Request, STATUS_CANCELLED);
    };
}
```

Функция обратного вызова для отмены запроса захватывает блокировку и устанавливает значение флага `IsCancelled` в `TRUE`, чтобы в случае параллельного исполнения на другом процессоре функции `EvtInterruptDpc` она не будет пытаться завершить запрос. Потом драйвер завершает аппаратную операцию ввода/вывода — но, не запрос ввода/вывода — вызывая `TerminateIo`. В случае успешного выполнения `TerminateIo` мы знаем, что функция `EvtInterruptDpc` не будет вызываться для завершения запроса. В результате драйвер может освободить дополнительную ссылку, полученную ранее функцией `EvtIoXXX`, и завершить запрос.

Неудачное же завершение исполнения `TerminateIo` означает, что операция ввода/вывода находится на грани завершения, и функция `EvtInterruptDpc` будет исполняться. Процедура обратного вызова для отмены запроса устанавливает значение флага `IsTerminateFailed` в `TRUE` и не завершает запрос, т. к. может быть, что аппаратура использует буферы в объекте запроса. Когда драйвер завершает запрос, инфраструктура завершит запрос.

шает низкоуровневый пакет IRP, таким образом освобождая буферы. Следовательно, если аппаратная операция ввода/вывода завершается, то запрос завершается функцией обратного вызова *EvtInterruptDpc*. А если аппаратная операция ввода/вывода не завершается, то запрос завершается функцией обратного вызова *EvtCancelCallback*. Если к буферам в объекте запроса не выполняется аппаратных обращений, то логику можно упростить, удалив флаг *IsTerminateFailed*.

5. В функции обратного вызова *EvtInterruptDpc* проверяем значение флагов *IsCancelled* и *IsTerminateFailed*, чтобы определить, завершать ли запрос.

```
EvtDpcForIsr(Interrupt)
{
    PREQUEST_CONTEXT reqContext = GetRequestContext (Request);
    NTSTATUS status;
    BOOLEAN completeRequest;
    KIRQL oldIrql;

    completeRequest = TRUE;
    KeAcquireSpinlock(&reqContext->Lock, &oldIrql);
    if (reqContext->IsCancelled == FALSE) {
        status = WdfRequestUnmarkCancelable(Request);
        if (status == STATUS_CANCELLED) {
            // Процедура отмены находится на грани вызова
            // или уже ожидает блокировку.
            // Позволим завершить запрос ей.
            completeRequest = FALSE;
        }
        // Обрабатываем запрос и завершаем его
        // после освобождения блокировки.
        //
        status = STATUS_SUCCESS;
    }
    else {
        // Процедура отмены уже исполнилась,
        // но, возможно, не завершила запрос.
        if (reqContext->IsTerminateFailed {
            status = STATUS_CANCELLED;
        }
        else {
            completeRequest = FALSE
        }
    }
    KeReleaseSpinlock(&reqContext->Lock, oldIrql);

    WdfObjectDereference(Request);
    if (completeRequest) {
        WdfRequestComplete(Request, status);
    };
}
```

В функции обратного вызова *EvtInterruptDpc* захватываем блокировку, после чего проверяем значение переменной *IsCancelled*. Если запрос не был отменен, помечаем его как неотменяемый. Если запрос отменится непосредственно перед тем, как он помечен как неотменяемый, метод *WdfRequestUnmarkCancelable* возвратит значение *STATUS\_CANCELLED*, а

инфраструктура вызовет функцию обратного вызова *EvtRequestCancel*. В таком случае необходимо позволить функции *EvtRequestCancel* завершить запрос. Если же запрос все еще не был отменен, то функция *EvtInterruptDpc* обрабатывает запрос, как положено, устанавливает значение *STATUS\_SUCCESS* и завершит запрос.

Значение *IsCancelled*, равное *TRUE*, говорит о том, что функция обратного вызова уже выполнилась; тем не менее необходимо проверить значение флага *IsTerminateFailed*, чтобы определить, завершила ли она запрос.

Значение флага *IsTerminateFailed*, равное *TRUE*, означает, что аппаратная операция ввода/вывода была завершена до отмены запроса, поэтому функция *EvtInterruptDpc* должна завершить запрос, возвращая значение *STATUS\_CANCELLED*. Значение этого флага, равное *FALSE*, предполагает, что запрос был завершен функцией обратного вызова для отмены.

Теперь драйвер может освободить блокировку и завершить запрос. Независимо от того, завершил ли драйвер запрос или нет, он всегда должен освобождать дополнительную ссылку, т. к. обратный вызов завершения не освобождает ее в случае неуспешного исполнения процедуры *TerminateIo*.

### Упростила ли WDF отмену запросов?

Ответ на этот вопрос зависит от того, каким образом драйвер удерживает длительные запросы, т. е. запросы, ожидающие аппаратное событие. Если для удержания таких запросов драйвер использует очереди, то тогда да, WDF упрощает процесс отмены. В противном же случае процесс запутанный.

При использовании очередей для удержания длительных запросов не требуется разбираться с какими-либо вопросами отмены. Любые запросы, находящиеся в очереди, будут завершены инфраструктурой при их отмене. Если необходимо, чтобы перед завершением запросов инфраструктура предоставляла соответствующее извещение, можно зарегистрировать обратный вызов функции *EvtIoCancelledOnQueue* и завершить запрос самостоятельно. В образце драйвера *Osrusbfx2* показано, каким образом парковать запросы, ожидающие аппаратное событие.

Если же для парковки длительных запросов вместо очереди используется собственная процедура отмены с применением метода *WdfRequestMarkCancelable*, то необходимо заботиться о возможном возникновении состояния гонок между процедурой отмены и асинхронным обратным вызовом (например, *EvtInterruptDpc*, *EvtTimerFunc* или *EvtWorkItem*), завершающим запрос. Необходимо обеспечить, что запросом однозначно владеет только один поток, прежде чем вызывать метод *WdfRequestComplete* для его завершения.

В качестве альтернативы этим двум подходам можно применить способ, в котором драйвер удерживает запрос в состоянии транзита, не помечает его как отменяемый, но все равно реагирует на отмену. Для этого драйвер периодически проверяет статус запроса, вызывая метод *WdfRequestIsCanceled*. Этот подход возможен, если выполняется активный опрос (polling) аппаратуры для выяснения статуса завершения аппаратной операции ввода/вывода. Но хотя этот подход кажется довольно простым, обычно он не является оптимальным, т. к. активный опрос в режиме ядра может вызвать падение производительности.

Удерживание запросов в очереди, предоставляемая инфраструктуре заботиться о вопросах отмены, является самым лучшим подходом. С этой точки зрения, да, WDF существенно упрощает процесс отмены запросов. (Ильяс Якуб (*Eliyas Yakub*), команда разработчиков *Windows Driver Foundation*, Microsoft.)

## Отмена входящих запросов синхронно с подзапросами

Некоторые драйверы создают и отправляют один или несколько подзапросов, чтобы сорвать данные, требуемые для завершения доставленного инфраструктурой запроса. При от-

мене доставленного инфраструктурой запроса, драйвер должен синхронизировать отмену его подзапросов.

Одним из способов синхронизации доступа при отмене является использование коллекции объектов запроса для отслеживания запросов, отправленных драйвером получателю ввода/вывода. Когда драйвер отправляет запрос, он добавляет дескриптор запроса в коллекцию, вызывая метод `WdfCollectionAdd`. Этот метод может завершиться неудачей, поэтому драйвер должен быть готовым обработать такую неудачу. Когда драйвер заканчивает обработку запроса, он удаляет дескриптор из коллекции.

Драйвер должен предохранять коллекцию с помощью типа блокировки, соответствующего уровню IRQL, на котором драйвер обращается к коллекции. В этом случае драйвер должен применять спин-блокировку, т. к. обратный вызов завершения ввода/вывода может исполняться на уровне DISPATCH\_LEVEL.

Для отмены запроса с помощью этого способа синхронизации драйвер должен выполнить следующие шаги:

1. Захватить блокировку для коллекции.
2. Найти в коллекции дескриптор объекта запроса.
3. Увеличить счетчик ссылок для объекта запроса.
4. Освободить блокировку.
5. Отменить запрос.
6. Уменьшить счетчик ссылок для объекта запроса.

Код в листинге 10.8, взятый из образца драйвера Usbsamp\Sys\Isorwr.c, является примером реализации драйвером этого типа синхронизации.

#### Листинг 10.8. Синхронизация отмены запроса в драйвере KMDF с помощью коллекции

```

WdfSpinLockAcquire(rwContext->SubRequestCollectionLock);
for(i =0; i < WdfCollectionGetCount(rwContext->
                                     SubRequestCollection); i++) {
    subRequest = (WDFREQUEST) WdfCollectionCetItem
                           (rwContext->SubRequestCollection, i);
    subReqContext = CetSubRequestContext(subRequest);
    WdfObjectReference(subRequest);
    InsertTailList(&cancelList, &subReqContext->ListEntry);
}
WdfSpinLockRelease(rwContext->SubRequestConnectionLock);

while(!IsEmptyList(&cancelList)) {
    thisEntry = RemoveHeadList(&cancelList);
    subReqContext = CONTAINING_RECORD(thisEntry,
                                       SUB_REQUEST_CONTEXT, ListEntry);
    subRequest = WdfObjectContextCetObject(subReqContext);
    if (!WdfRequestCancelSentRequest(subRequest)) {
        . . . // Код для обработки ошибок опущен.
    }
    WdfObjectDereference(subRequest);
}

```

В листинге 10.8 драйвер разбивает входящие запросы ввода/вывода на меньшие подзапросы, которые он отправляет получателю ввода/вывода. Каждый подзапрос является объектом типа `WDFREQUEST` и имеет область контекста объекта, в которой содержится дескриптор основного запроса и поле `LIST_ENTRY`, а также другие специфичные для запроса данные.

Если по какой-либо причине входящий запрос отменяется, то драйвер также должен отменить и ассоциированные с ним подзапросы. Драйвер помечает основной запрос как отменяемый и создает объект коллекции, в который он помещает подзапросы. При каждом завершении подзапроса драйвер удаляет его из коллекции.

Код, приведенный в листинге, является частью функции обратного вызова `EvtRequestCancel` для главного запроса. Для защиты коллекции драйвер применяет спин-блокировку, т. к. инфраструктура может активировать обратный вызов завершения ввода/вывода на уровне `DISPATCH_LEVEL`.

Если основной запрос отменяется, то драйвер захватывает блокировку для коллекции и проходит в цикле по ее элементам, чтобы создать связанный список всех подзапросов и получить дополнительную ссылку на каждый подзапрос при его добавлении в список. Вместо того чтобы помещать в список весь объект подзапроса, драйвер просто связывает их области контекста посредством поля `LIST_ENTRY`. По завершению создания списка драйвер освобождает блокировку.

## Сводка методов синхронизации и общие советы

В табл. 10.4 приведена сводка способов синхронизации, которыми могут пользоваться драйверы WDF.

**Таблица 10.4. Сводка методов синхронизации доступа**

Способ синхронизации	Использование в UMDF	Использование в KMDF
Созданная инфраструктурой блокировка представления объекта для объектов устройств и очередей	Захватывается и освобождается методами <code>IWDFObject::AcquireLock</code> и <code>IWDFObject::ReleaseLock</code> , соответственно, на объекте устройства или очереди	Захватывается и освобождается методами <code>WdfObjectAcquireLock</code> и <code>WdfObjectReleaseLock</code> соответственно
Область синхронизации	Конфигурируется методом <code>IWDFDeviceInitialize::SetLockingConstraint</code>	Конфигурируется в структуре атрибутов объекта драйвера, устройства, очереди и файла
Автоматическая сериализация	Неприменимо	Конфигурируется в структуре конфигурации объекта для процедур DPC, таймеров и рабочих элементов
Уровень исполнения	Неприменимо	Конфигурируется в структуре атрибутов объекта драйвера, устройства, файла и общих объектов
Wait-блокировка	Неприменимо; используются критические секции Windows	Создается и манипулируется с помощью методов семейства <code>WdfWaitLockXxx</code>
Спин-блокировки	Неприменимо; для драйверов UMDF не требуется синхронизация на высоких уровнях IRQL	Создается и манипулируется с помощью методов семейства <code>WdfSpinLockXxx</code>

Таблица 10.4 (окончание)

Способ синхронизации	Использование в UMDF	Использование в KMDF
События, мьютексы и другие механизмы, определенные в Windows	Используются функции Windows API	Используются функции интерфейса DDI режима ядра Windows
Interlocked-функции	Используются функции Windows API	Используются функции интерфейса DDI режима ядра Windows

Далее приводятся несколько общих рекомендаций для реализации синхронизации доступа в драйверах WDF.

- ◆ Используйте возможности синхронизации доступа, предоставляемые WDF, для объектов и обратных вызовов WDF. Для выполнения арифметических или логических операций с переменной или для координирования действий драйвера с иными, нежели WDF, возможностями и драйверами используйте механизмы Windows.
- ◆ При разработке очередей ввода/вывода для драйвера следует принимать в учет способности устройства и установить метод диспетчеризации и область синхронизации, позволяющие получить требуемый уровень параллельности.
- ◆ Следует иметь четкое понимание типов данных, совместно используемых функциями обратного вызова драйвера, а также уметь различать между совместным доступом только для чтения и совместным доступом как для чтения, так и для записи. Применяйте блокировки только в том случае, когда доступ не сериализуется методом диспетчеризации очереди и областью синхронизации, предоставляемым драйвером.
- ◆ Не применяйте блокировки только с целью предотвращения удаления объекта. Вместо этого получите ссылку и предоставьте управление временем жизни инфраструктуре.
- ◆ Чтобы избежать взаимоблокировок, если драйвер использует несколько блокировок, необходимо всегда захватывать их в одном и том же порядке, а освобождать в порядке, обратном порядку захвата. То есть, если драйвер захватывает блокировку А перед блокировкой В, он должен освободить блокировку В перед блокировкой А.
- ◆ Для драйверов KMDF следует понимать уровни IRQL, на которых исполняются отдельные функции.

# ГЛАВА 11

## Трассировка и диагностируемость драйверов

Трассировка программного обеспечения заключается во вставке в различных точках кода драйвера операторов для генерирования сообщений трассировки с целью регистрации его поведения.

Сообщения можно потом просматривать в режиме реального времени при исполнении драйвера или сохранить в файле регистраций для просмотра позже. Трассировка обычно применяется для определения местонахождения и причины ошибок, но с ее помощью можно также выполнять другие задачи, например, анализ частоты вызова разных процедур.

Трассировка в драйверах обычно основана на подсистеме уровня ядра Event Tracing for Windows (Трассировка событий Windows), которая записывает в журнал сообщения трассировки для процессов режима ядра и пользовательского режима. Так как обычно использование ETW может быть сопряжено с некоторыми трудностями, большинство разработчиков драйверов используют препроцессор WPP (Windows Preprocessor), что упрощает и улучшает процесс оснащения драйвера для трассировки с применением ETW.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b> Инструменты трассировки WDK Шаблонные файлы KMDF	%wdk%\tools\tracing\%wdk%\bin\wppconfig
<b>Образцы драйверов</b> Evntdrv Fx2_Driver Osrusbfx2	%wdk%\src\general\evntdrv %wdk%\src\umdf\usb\fx_2driver %wdk%\src\kmdf\toaster
<b>Документация WDK</b> WPP Software Tracing <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80090">http://go.microsoft.com/fwlink/?LinkId=80090</a>
<b>Прочее</b> Раздел "Event Tracing" в наборе средств разработки Platform SDK	<a href="http://go.microsoft.com/fwlink/?LinkId=84477">http://go.microsoft.com/fwlink/?LinkId=84477</a>

<sup>1</sup> Программная трассировка с применением WPP. — Пер.

## Основы программной трассировки с применением WPP

Программная трассировка является одним из самых давних способов отладки. Она предоставляет удобный и гибкий способ для получения подробной информации о поведении исполняющегося драйвера в ключевых точках его работы. Программная трассировка создает журнал учета работы драйвера. С помощью сообщений трассировки в этом журнале можно сохранить различную полезную информацию. Например, такую:

- ◆ причины изменений состояний;
- ◆ извещения о сборке мусора или операций очистки;
- ◆ выделения памяти;
- ◆ события ввода/вывода, например завершение или отмена запросов;
- ◆ данные об ошибках.

В этой главе рассматривается реализация трассировки WPP в драйвере WDF для создания журнала трассировки, содержащего историю событий драйвера. Родственным источником сообщений трассировки является журнал KMDF, который основан на WPP.

Дополнительная информация о журнале KMDF приводится в [главе 22](#).

### Преимущества программной трассировки WPP

Существует много способов для осуществления программной трассировки. Программная трассировка WPP предоставляет следующие важные преимущества над другими методами, такими как, например, операторы печати отладчика.

#### Гибкое динамическое управление

Функциональность создания сообщений можно включать и выключать в исполняющемся драйвере. Для этого не требуется останавливать и перезапускать драйвер или перезагружать операционную систему. Сообщения трассировки можно пометить флагами, чтобы выводились только определенные выбранные, а не все возможные сообщения. Кроме этого, в один и тот же журнал трассировки можно записывать сообщения из нескольких источников. Например, можно выбрать, чтобы выводились сообщения от драйвера в порядке их возникновения и связанные компоненты операционной системы.

#### Возможность просмотра сообщений в режиме реального времени или запись их в файл

В Windows XP и последующих версиях Windows сообщения трассировки можно просматривать в режиме реального времени. Можно также направить сообщения в журнал трассировки для просмотра позже. Журнал может быть расположен даже совсем на другом компьютере.

#### Обильная информация

Сообщения трассировки WPP могут содержать практически любые полезные данные, включая информацию о текущей конфигурации драйвера или значения ключевых переменных. При компоновке проекта препроцессор WPP добавляет к каждому сообщению трассировки

имя функции, имя исходного файла и номер строки. Когда генерируется сообщение трассировки, регистрационный механизм ETW автоматически прикрепляет к нему временную отметку.

## Охрана интеллектуальной собственности

Сообщение трассировки WPP издаются в двоичном формате. Так как информация о форматировании сохраняется отдельно от двоичного файла драйвера и обычно не поставляется вместе с продуктом, то сторонним лицам будет трудно извлечь сведения, представляющие интеллектуальную собственность, из сообщений трассировки или воспользоваться этими сообщениями для обратного инжиниринга драйвера.

## Легкость перехода от операторов печати отладчика

Для драйверов, в которых для трассировки в настоящее время применяются операторы печати отладчика, WPP может преобразовать вызовы операторов печати в сообщения трассировки в процессе компоновки. Таким образом, существующая трассировочная оснастка преобразовывается в более эффективную форму, без необходимости выполнять какие бы то ни было изменения в коде.

## Включение в поставляемые продукты

Программную трассировку WPP можно включить как в проверочную, так и в свободную сборку продукта, поэтому код трассировки можно оставить в двоичных файлах продукта, поставленного клиенту. Это позволяет применять инструменты трассировки WDK на выездах для просмотра сообщений трассировки на драйверах, установленных на системах клиента.

## Минимальное воздействие на производительность

Вызовы сообщений трассировки WPP исполняются только тогда, когда трассировка явно разрешена внешним оператором. Если трассировка не включена, то код трассировки никогда не вызывается и не оказывает никакого влияния на производительность. Но даже при включенной трассировке ее влияние на производительность является минимальной, т. к. сообщения трассировки издаются в двоичном формате. Трудоемкая задача форматирования и вывода сообщений на экран выполняется отдельным приложением. Трассировка особенно полезна для исследования причины ошибок, чувствительных к производительности системы<sup>1</sup>. Эти ошибки часто не проявляются при попытках наблюдения за поведением драйвера с использованием таких методов, как операторы печати отладчика, которые могут значительно замедлить скорость исполнения драйвера.

## Компоненты программной трассировки WPP

Программная трассировка WPP состоит из нескольких компонентов. В этом разделе приводится краткое описание каждого компонента и показано взаимодействие компонентов в сеансе трассировки.

<sup>1</sup> Такая ошибка еще называется гейзенбаг или гейзенберговская ошибка — системная ошибка, которая — по аналогии с принципом неопределенности Гейзенberга в квантовой физике — исчезает или видоизменяется при попытке ее выявления. — Пер.

## Поставщик трассировки

Поставщиком трассировки может быть любое приложение, компонент операционной системы или драйвер, оснащенный для трассировки с применением WPP. Для трассировки конкретных операций драйвер можно разбить на несколько поставщиков трассировки, даже если драйвер содержитя в одном исходном файле. В этой главе рассматривается, как оснастить драйвер WDF в качестве поставщика трассировки.

## Контроллер трассировки

Поставщик трассировки генерирует сообщения трассировки только тогда, когда он включен контроллером, которым является приложение или инструмент, управляющий сеансом трассировки. Два широко используемых контроллера трассировки — TraceView и TraceLog — включены в инструментарий WDK. Контроллер трассировки может включить несколько поставщиков для одного сеанса, в том числе поставщиков от разных драйверов или системных компонентов. В сессии с несколькими поставщиками сообщения от всех поставщиков записываются в журнал впемерешку в едином порядке их поступления. Если в драйвере определено несколько поставщиков, то контроллер трассировки можно настроить для пропуска сообщений только от определенных поставщиков. Но при трассировке с применением WPP, разных поставщиков от одного драйвера можно включить только для одного сеанса.

Описание создания своего собственного контроллера трассировок с помощью ETW API приводится в разделе **Controlling Event Tracing Sessions** (Управление сеансом трассировки событий) в документации набора средств разработки Platform SDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80062>.

## Буфер трассировки

Система содержит набор буферов для хранения сообщений трассировки для каждого сеанса трассировки. Контроллер трассировок может сконфигурировать размер этих буферов для сеанса. Буфера автоматически сбрасываются в журнал трассировок или потребителю трассировок через определенный интервал времени или когда они полностью заполняются. Буфера также автоматически сбрасываются, когда контроллер трассировок останавливает сеанс трассировки. Кроме этого, контроллер трассировок может явно сбрасывать буфера или по требованию, или через определенные интервалы времени. Крах системы не затрагивает буфера, и сообщения трассировки не теряются.

## Сеанс трассировки

Сеанс трассировки — это период времени, в течение которого включены один или несколько поставщиков трассировки, порождающие последовательность сообщений трассировки, которая записывается в буфер. Эта последовательность сообщений называется *журналом трассировок*. Контроллер трассировки запускает и конфигурирует сеанс, включает одного или несколько поставщиков и ассоциирует их с сеансом. В сессии с несколькими поставщиками сообщения от всех поставщиков записываются в журнал трассировки вперемешку, в едином порядке их поступления. Во время сеанса контроллер трассировки может опрашивать свойства сеанса и обновлять их, а также остановить сеанс.

### Совет

В WPP поставщика трассировок можно ассоциировать только с одним сеансом трассировок. Если контроллер включает этот поставщик в другом сеансе, то он отключается в первоначальном сеансе.

Если сеанс трассировки ассоциирован с регистратором ядра Windows (kernel logger), то в журнал трассировок включаются предопределенные системные события, порождаемые операционной системой, например события ввода/вывода диска или события страничной ошибки. Добавив в сеанс поставщика от ядра, можно получить один журнал трассировок, содержащий вперемешку сообщения трассировки, как от драйвера, так и от ядра. В этом журнале фиксируются события ядра Windows по отношению к событиям драйвера.

## Потребитель трассировки

*Потребитель трассировки* — это приложение или инструмент, который получает, форматирует и выводит на экран содержимое журнала трассировки. Потребитель трассировки форматирует сообщения трассировки в удобочитаемый формат. Для этого применяются инструкции либо из файла идентификаторов базы данных программы поставщика (расширение файла — PDB), либо из отдельного файла форматирования сообщений трассировки (расширение файла — TMF). Часто потребитель трассировочных данных также функционирует в качестве контроллера трассировки.

Взаимодействие потребителя с другими компонентами трассировки в типичной системе показано на рис. 11.1.

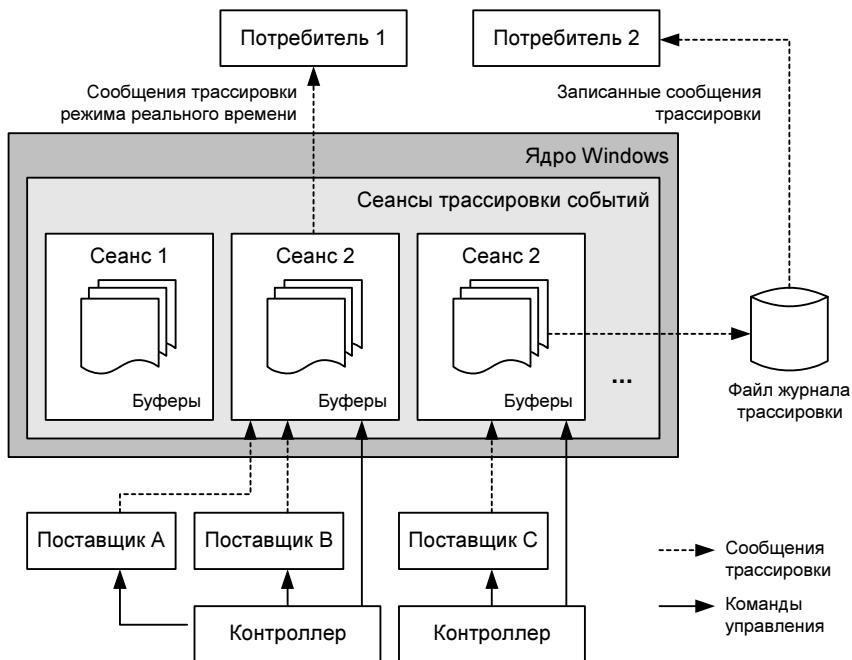


Рис. 11.1. Архитектура программной трассировки

Потребитель трассировки можно использовать для просмотра журнала трассировки двумя разными способами:

- ◆ дать указание контроллеру трассировки передавать сообщения из буферов сеанса трассировки прямо потребителю трассировочных данных, после чего выводить их для просмотра в режиме реального времени;

- ◆ дать указание контроллеру трассировок записывать сообщения в файл журнала трассировки (расширение файла — ETL) для данного сеанса. Сообщения, сохраненные в файл, можно прочитать позже, используя потребителя трассировки, возможно даже, совсем на другом компьютере.

На рис. 11.1 показаны три активных сеанса блокировки:

- ◆ у сеанса 1 нет ни поставщиков, ни потребителей, ассоциированных с ним в настоящее время. Их наличие не является обязательным условием для существования сеанса;
- ◆ сеанс 2 имеет два ассоциированных с ним поставщика — А и В; потребитель 1 получает сообщения в перемешку от обоих поставщиков в режиме реального времени;
- ◆ сеанс 3 имеет одного поставщика — С; сообщения в нем записываются в файл журнала трассировки и просматриваются потребителем 2 после окончания сеанса.

## WPP и ETW

Базовая архитектура средства программной трассировки WPP основана на технологии ETW. С помощью ETW можно организовать мощную и гибкую трассировку, но в то же самое время реализация такой трассировки будет сопряжена со сложностями. Средство WPP было создано с целью упростить и улучшить возможности трассировки, предоставляемые ETW, особенно для драйверов, которым требуется только простая трассировка.

Принцип работы WPP состоит в следующем:

1. Макросы WPP для конфигурирования трассировки и генерирования сообщений трассировки включаются в исходный код драйвера.
2. Файл Sources проекта дает указание утилите Build выполнить препроцессор WPP перед первым проходом компилятора. Препроцессор преобразовывает макросы WPP и файлы шаблонов в код, требуемый для реализации непосредственной трассировки ETW.
3. Код, сгенерированный WPP для каждого исходного файла, сохраняется в заголовочном файле сообщений трассировок (с расширением TMH). Этот файл также содержит макросы, которые добавляют инструкции для форматирования сообщений трассировки для файла идентификаторов драйвера (с расширением PDB).
4. Каждый исходный файл содержит ассоциированный с ним TMH-файл. При компиляции исходного кода, код в TMH-файле компилируется вместе с ним точно так же, как и обычный заголовочный файл.

## ETW в Windows Vista

Начиная с Windows Vista, операционная система предоставляет единый формат событий, что позволяет приложениям протоколировать систематизированные события для инструментов, автоматического мониторинга и диагностических компонентов. Windows Vista ETW API протоколирует эти события в ETW. Этот формат событий определяет события, используя файл манифеста формата XML, который позже компилируется как ресурс и прикрепляется к двоичному файлу драйвера. Каждое событие имеет стандартные информационные метаданные и изменяемый раздел полезных данных, который может содержать набор простых типов или структуры данных, такие как массивы или структуры. С помощью этой информации потребители сообщений трассировки могут декодировать событие, после того как оно запротоколировано в ETW.

## Полезная информация

WPP не использует Windows Vista ETW API, а вместо этого протоколирует события, применяя более раннюю версию ETW API. Чтобы использовать новые возможности, предоставляемые Windows Vista, трассировку необходимо реализовывать с помощью Windows Vista ETW API.

Протоколирование событий в таком формате удобно для компонентов, использующих журналы событий. Такие компоненты, как сервис EventLog и утилита Event Viewer, могут без труда использовать эти систематизированные события для реализации возможностей обнаруживаемости, выполнения опросов и фильтрации, которые позволяют пользователям быстро сфокусироваться на представляющих интерес событиях. Применение систематизированного формата также позволяет локализировать сообщения о событиях и другие типы сообщений.

Кроме этого, самовосстанавливающиеся компоненты, такие как инфраструктура Windows Diagnostic Infrastructure (Диагностическая инфраструктура Windows), используют систематизированный формат событий для мониторинга событий, связанных с конкретными проблемами.

При возникновении таких событий, если указан модуль решений, инфраструктура WDI может автоматически исправить проблему.

Описание Windows Vista ETW API приводится в разделе **Adding Event Tracing to Kernel-Mode Drivers** (Добавление возможности трассировки событий в драйверы режима ядра) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80610>. Также, для примера реализации трассировки, используя Windows Vista ETW API, см. образец драйвера Evntdrv в WDK, который находится по адресу %wdk%\src\general\evntdrv.

## Функции и макросы для работы с сообщениями трассировки

Центральным компонентом в любой реализации трассировки является функция, которая генерирует сами сообщения трассировки. Функция генерации сообщения трассировки вставляется в необходимое место в коде и конфигурируется, чтобы помещать полезную информацию в сообщение. Генерируемое функцией сообщение может содержать любую полезную информацию, обычно какой-либо текст с содержанием сообщения и значения любых представляющих интерес переменных. Существуют три основных способа генерации сообщений трассировки WPP.

- ◆ С помощью макроса `DoTraceMessage`. Это способ по умолчанию.

Препроцессор WPP преобразует макрос `DoTraceMessage` в вызов соответствующей функции ETW. Какая именно функция ETW применяется, зависит от версии Windows, для которой компилируется драйвер, а также от режима исполнения драйвера — пользовательского или ядра.

- ◆ Через преобразование операторов печати отладчика в сообщения трассировки.

Препроцессору WPP можно дать указание преобразовывать существующие операторы печати отладчика в эквивалентные функции сообщений трассировки.

- ◆ С помощью специальных функций сообщений трассировки.

Эти функции полезны в том случае, когда макрос `DoTraceMessage` не поддерживает требования разрабатываемого драйвера. Примеры специальных функций сообщений трассировки приводятся далее в этой главе.

### Внимание!

Необходимо соблюдать осторожность, чтобы не раскрыть в сообщениях трассировки данные, относящиеся к системе безопасности, или личные данные пользователей. Любой потребитель трассировки, который может получить доступ к ТМН-файлам или РДВ драйвера, может отформатировать и прочитать журнал трассировки драйвера.

## Макрос *DoTraceMessage*

Макрос *DoTraceMessage* является стандартным способом генерирования сообщений трассировки. Формат макроса следующий:

*DoTraceMessage (Flag, Message, MessageVariables...)*

Здесь:

- ◆ *Flag* — любой из определенных пользователем флагов трассировки, который потребители трассировки могут использовать для указания, какие сообщения трассировки нужно генерировать. Примеры, как определять и использовать эти флаги, приводятся в разд. "Поддержка программной трассировки в драйвере" далее в этой главе;
- ◆ *Message* — строковая константа, определяющая формат сообщения трассировки, подобно строке формата функции *printf*. В строке можно использовать стандартные спецификаторы форматы функции *printf*, такие как %d и %s, и текст. WPP также поддерживает специальные форматы, такие как %!NTSTATUS! и %!HRESULT!, которые выводят на экран строчные эквиваленты значений статуса режима ядра и значения *HRESULT* пользовательского режима соответственно;
- ◆ *MessageVariables* — список разделенных запятыми переменных, чьи значения добавляются к строке сообщения в соответствии с форматом, указанным в параметре *Message*.

Подробную информацию см. в разделе *DoTraceMessage* на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80611>.

## Преобразование операторов печати отладчика в ETW

Чтобы дать указание WPP преобразовывать операторы печати отладчика, необходимо вставить следующую опцию в директиву *RUN\_WPP* в файле Sources проекта:

*-func:DebugPrintFunction(LEVEL, MSG, ...)*

Здесь *DebugPrintFunction* — это имя функции печати отладчика. Эта директива дает указание препроцессору WPP заменить все случаи *DebugPrintFunction* эквивалентными макросами WPP.

Директива *RUN\_WPP* рассматривается в разд. "Поддержка программной трассировки в драйвере" далее в этой главе.

## Условия для сообщений

Большинство функций сообщений трассировки позволяет указать одно или несколько условий для сообщения. Например, с помощью условия можно указать, является ли сообщение трассировки просто информационным или же оно связано с предупреждением либо ошибкой. Для макроса *DoTraceMessage* существует только одно условие, которое принимает определенные пользователем флаги. Можно также реализовать специальную функцию сообщения трассировки, использующую другое условие или несколько условий.

С помощью условий сообщения разбиваются на категории. Потребитель трассировки может сконфигурировать сеанс таким образом, чтобы ассоциированные поставщики трассировки генерировали сообщения только из выбранных категорий. Эта возможность позволяет потребителю трассировки увеличить отношение полезных сообщений к бесполезным в журнале трассировки, разрешая протоколирование только определенного подмножества сообщений трассировки.

Например, если условие сообщения принимает стандартные уровни трассировки, как определено в файле Eventtrace.h, потребитель трассировки может сконфигурировать сеанс для вывода только сообщения ошибок и предупреждений и исключить информационные сообщения. Специальная функция сообщения трассировки может иметь дополнительное условие, указывающее часть драйвера, из которой происходит сообщение. Потребитель трассировки может использовать эти условия для вывода только те сообщения об ошибках, ассоциированные с запросами IOCTL.

## Специальные функции сообщений трассировки

Если макрос `DoTraceMessage` не удовлетворяет требованиям драйвера, можно определить одну или несколько специальных функций сообщений трассировки. Общий формат, которого должна придерживаться специальная функция сообщения трассировки, подобен тому, который используется в макросе `DoTraceMessage`. Отличие состоит в том, что специальные функции сообщений трассировки могут поддерживать разные условия, включая множественные условия, как показано в следующем примере:

```
FunctionName(Condition1, Condition2, ..., "Message", MessageVariables...)
```

Значения параметров специальной функции сообщения трассировки следующие:

- ◆ `Condition1, Condition2` — одно или несколько условий, разделенных запятыми. Все параметры, идущие перед сообщением, рассматриваются как условия. Сообщение трассировки генерируется, только если выполняются все условия;
- ◆ `Message` — строковая константа, определяющая формат сообщения трассировки. Спецификаторы формата такие же, как и для макроса `DoTraceMessage`;
- ◆ `MessageVariables` — список разделенных запятыми определенных драйвером переменных, чьи значения добавляются к сообщению в соответствии с форматом, указанным в параметре `Message`.

Чтобы использовать специальную функцию сообщения трассировки, необходимо сконфигурировать проект таким образом, чтобы препроцессор WPP распознавал специальную функцию и правильно обрабатывал вывод. Для примера реализации специальных функций сообщений трассировки см. разд. "Пример UMDF: директива `RUN_WPP` для образца драйвера `Fx2_Driver`" далее в этой главе.

## Поддержка программной трассировки в драйвере

Чтобы оснастить драйвер для программной трассировки, необходимо выполнить следующие шаги:

1. Модифицировать файл Sources для исполнения препроцессора WPP.
2. Подключить TMH-файлы.

3. Определить контрольный GUID и флаги сообщений трассировки.
4. Добавить макросы для инициализации и очистки трассировки.
5. Оснастить код драйвера для генерирования сообщений трассировки в соответствующих точках.

В этом разделе рассматривается механика для реализации программной трассировки WPP в драйверах WDF. По существу, одни и те же процедуры требуются как для драйверов UMDF, так и для драйверов KMDF, хотя некоторые из макросов WPP определены по-разному для режима ядра и пользовательского режима.

## Модификации файла Sources для исполнения препроцессора WPP

Хотя WPP встроена в среду компоновки WDK, утилите компоновки WDK необходимо дать явное указание на исполнение препроцессора WPP. Для этого в конец файла Source драйвера добавляется директива `RUN_WPP`. Эта директива также используется для указания специальных функций сообщений трассировки. Формат директивы `RUN_WPP` следующий:

```
RUN_WPP= $(SOURCES) Option1 Option2 ...
```

Используемые опции зависят от типа драйвера, UMDF или KMDF, и каким образом реализована трассировка в драйвере. В табл. 11.1 приводится сводка опций директивы `RUN_WPP`, наиболее полезных для драйверов KMDF и UMDF.

**Таблица 11.1. Опции для директивы `RUN_WPP`**

Опция	Описание
<code>-km</code>	Требуется для драйверов KMDF. По умолчанию трассируются только компоненты пользовательского режима. Эта опция определяет макрос <code>WPP_KERNEL_MODE</code> , который выполняет трассировку компонентов режима ядра
<code>-func: (funcname (param1, param2, ...))</code>	Необязательная. Определяет специальную функцию трассировки. Этот параметр можно повторять для каждой такой функции
<code>-scan:filename</code>	Необязательная. Сканирует параметр <code>filename</code> , чтобы найти определения специальной функции трассировки. Формат этого определения приводится в следующем разделе
<code>-dll</code>	Требуется для драйверов UMDF; не используется для драйверов KMDF. Определяет макрос <code>WPP_DLL</code> , который вызывает инициализацию структур данных WPP при каждом вызове макроса <code>WPP_INIT_TRACING</code> . В противном случае структуры инициализируются только один раз
<code>-gen:{filename.tpl}* .tmh</code>	Необязательная. Дает указание препроцессору WPP создать TMH-файл из шаблона с именем <code>filename.tpl</code> . Этот шаблон применяется только для драйверов KMDF

### Полезная информация

Опции `-func` и `-scan` применяются только для проектов, имеющих одну или несколько специальных функций сообщений трассировки. Если для генерации сообщений применяется только макрос `DoTraceMessage`, то эти опции не используются. Для специальных функций

сообщений трассировки опции `-func` и `-scan` являются необязательными только в том смысле, что необходимо использовать одну или другую из них, чтобы предоставить WPP прототип функции.

В следующих двух разделах показано конфигурирование директивы `RUN_WPP` на примерах драйверов `Fx2_Driver` и `Osrusbf2`.

Дополнительную информацию об опциях для директивы `RUN_WPP` см. в разделе **WPP Software Tracing** в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80090>.

### **Пример UMDF: директива `RUN_WPP` для образца драйвера `Fx2_Driver`**

Директива `RUN_WPP` для файла Sources образца драйвера `Fx2_Driver` имеет следующий формат:

```
RUN_WPP= $(SOURCES) -dll -scan:internal.h
```

Так как это драйвер UMDF, необходимо установить опцию `-dll`. Специальные функции сообщений трассировки для драйвера `Fx2_Driver` определены в заголовочном файле `Internal.h`. В директиве `RUN_WPP` опция `-scan` используется, чтобы указать препроцессору WPP на файл `Internal.h`, содержащий определения функций, которые показаны в листинге 11.1.

#### **Листинг 11.1. Определения функций для специальных сообщений трассировки**

```
// begin_wpp config
// FUNC Trace{FLAC=MYDRIVER_ALL_INFO} (LEVEL, MSC, ...);
// FUNC TraceEvents (LEVEL, FLAGS, MSC, ...);
// end_wpp
```

Определения специальных сообщений трассировки имеют следующие характеристики:

- ◆ все элементы определения закомментированы, чтобы предотвратить попытки компилятора выполнять над ними синтаксический анализ;
- ◆ блок определений начинается оператором `begin_wpp config` и заканчивается оператором `end_wpp`;
- ◆ определение каждой функции начинается с ключевого слова `FUNC`, за которым следует имя функции;
- ◆ после имени функции следует определение входных параметров.

Подробную информацию о формате определений сообщений трассировки см. в разделе **Can I customize DoTraceMessage?** (Можно ли переделать под свои требования макрос `DoTraceMessage`) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80061>.

### **Специальное сообщение трассировки драйвера `Fx2_Driver`**

В драйвере `Fx2_Driver` определена функция специального сообщения трассировки `TraceEvents`, которая поддерживает следующие условия:

- ◆ `LEVEL` — принимает стандартные уровни трассировки;
- ◆ `FLAG` — имеет четыре определенные пользователем значения для указания компонента драйвера, сгенерировавшего сообщение.

Значения условия `FLAG` определены в макросе `WPP_CONTROL_GUIDS`.

При определении функций специальных сообщений трассировки также необходимо определить специальную версию следующих макросов WPP:

- ◆ `WPP_LEVEL_LOGGER` — определяет сеанс, включивший драйвер в качестве поставщика и возвращает дескриптор сеанса;
- ◆ `WPP_LEVEL_ENABLED` — определяет, включено ли протоколирование для определенного значения флага.

Имена специализированных версий макросов должны отвечать требованиям стандартного формата и включать условия, поддерживаемые функцией специального сообщения трассировки. Общий формат имен макросов следующий:

- ◆ `WPP_CONDITIONS_ENABLED` — заменяет `WPP_LEVEL_ENABLED`;
- ◆ `WPP_CONDITIONS_LOGGER` — заменяет `WPP_LEVEL_LOGGER`.

`CONDITIONS` — это список условий, поддерживаемых функцией, в том порядке, в каком они приводятся в списке параметров функции, разделенные символами подчеркивания. Например, в функции специального сообщения трассировки, поддерживающей условия `LEVEL` и `FLAG`, модифицированная версия макроса `WPP_LEVEL_ENABLED` должна называться `WPP_LEVEL_FLAGS_ENABLED`.

Модифицированные версии этих макросов обычно определяются на основе стандартных версий. Для большинства драйверов необходимо предоставить определение значительного объема только для макроса `WPP_CONDITIONS_ENABLED`. Макрос `WPP_CONDITIONS_LOGGER` обычно можно просто установить в `WPP_LEVEL_LOGGER`.

Макросы для функции `TraceEvents` определяются в заголовочном файле `Internal.h` (листинг 11.2).

#### Листинг 11.2. Макросы WPP для драйвера `Fx2_Driver`

```
#define WPP_LEVEL_FLACS_LOCCKER(lvl, flags) \
    WPP_LEVEL_LOCCKER(flags)
#define WPP_LEVEL_FLACS_ENABLED(lvl, flags) \
    (WPP_LEVEL_ENABLED(flags) && WPP_CONTROL(WPP_BIT_## \
        flags).Level >= lvl)
```

В макросах порядок параметров функции `TraceEvents` определяется с параметром `LEVEL` на первом месте, а после него параметром флагов `FLAG`. Определения следующие:

- ◆ `WPP_LEVEL_FLAGS_LOGGER` — идентичный макросу `WPP_LEVEL_LOGGER`;
- ◆ `WPP_LEVEL_FLAGS_ENABLED` — возвращает `TRUE`, если включено протоколирование для определенного значения `FLAG`, и значение разрешенного уровня `LEVEL` больше, чем значение параметра `Level`.

#### Пример KMDF: директива `RUN_WPP` для образца драйвера `Osrusbf2`

В образце драйвера `Osrusbf2` также применяется специальная функция сообщения трассировки `TraceEvents`. Это сообщение было подробно рассмотрено в разд. "Специальное сообщение трассировки драйвера `Fx2_Driver`" ранее в этой главе. В следующем примере (листинг 11.3) приводится директивы `RUN_WPP` для файла `Sources` образца драйвера `Osrusbf2`.

**Листинг 11.3. Директива RUN\_WPP для драйвера Osrusbf2**

```
RUN_WPP = $(SOURCES)
        -km                                \
        -func:TraceEvents(LEVEL,FLAGS,MSK,...) \
```

В примере директива разбита на несколько строк с помощью символа обратной косой черты (\). Директива содержит две опции:

- ◆ -km — указывает, что драйвер исполняется в режиме ядра;
- ◆ -func — задает одну специальную функцию сообщения TraceEvents, которая поддерживает условия LEVEL и FLAG.

**Полезная информация**

Исходный код драйвера Osrusbf2 также содержит определение функции TraceEvents в заголовочном файле Trace.h. Трассировка в драйвере Osrusbf2 разрешена, только если значение EVENT\_TRACING определено как TRUE. Блок кода с определением функции TraceEvents компилируется только при условии, что значение EVENT\_TRACING не установлено в TRUE. Целью здесь является предоставить определение по умолчанию функции TraceEvents, когда трассировка не разрешена и препроцессор WPP не создает функцию TraceEvents.

В файле Sources также определяются специальные версии макросов WPP\_LEVEL\_LOGGER и WPP\_LEVEL\_ENABLED, рассмотренных ранее.

**Подключение ТМН-файла**

Каждый исходный файл, содержащий макросы WPP, должен подключать соответствующий ТМН-файл. Этот файл содержит код, генерируемый препроцессором WPP, и его нужно компилировать вместе с соответствующим исходным файлом. ТМН-файл должен иметь такое же имя, как и соответствующий исходный файл, но с расширением TMH. Например, ТМН-файл для исходного файла Driver.c называется Driver.tmh, а директива #include для его включения выглядит следующим образом:

```
#include driver.tmh
```

Файл ТМН должен быть подключен после макроса WPP\_CONTROL\_GUIDS, но перед вызовами любых других макросов WPP.

**Пример UMDF: подключение ТМН-файла**

Для драйвера Fx2\_Driver макрос WPP\_CONTROL\_GUIDS находится в заголовочном файле Internal.h. Чтобы обеспечить, что макрос WPP\_CONTROL\_GUIDS будет первым, в исходных файлах заголовочный файл Internal.h подключается перед ТМН-файлом. Например, в исходном файле ReadWriteQueue.cpp эти два файла подключаются так, как показано в листинге 11.4.

**Листинг 11.4. Подключение ТМН-файла в файле ReadWriteQueue.cpp для драйвера Fx2\_Driver**

```
#include "internal.h"
#include "ReadWriteQueue.tmh"
```

**Пример KMDF: подключение ТМН-файла**

В драйвере Osrusbf2 определяется контрольный GUID и флаги трассировки в заголовочном файле Trace.h, который в свою очередь подключается в заголовочном файле Osrusbf2.h.

Соответственно, в исходном файле этого драйвера файл Osrusbf2.h подключается перед TMH-файлом, как это делается в файле Interrupt.c в листинге 11.5.

#### Листинг 11.5. Подключение TMH-файла в файле Interrupt.c для драйвера Osrusbf2

```
#include <osrusbf2.h>
#if defined(EVENT_TRACINC)
#include "interrupt.tmh"
#endif
```

В примере KMDF TMH-файл подключается только тогда, когда включена трассировка событий. В примере в файле Sources определяется переменная среды EVENT\_TRACING вместе с переменной ENABLE\_EVENT\_TRACING. Когда определяется переменная ENABLE\_EVENT\_TRACING, файл Sources также определяет переменную EVENT\_TRACING и подключает директиву RUN\_WPP, которая активирует препроцессор WPP. С помощью этого приема можно включить или исключить трассировки WPP при сборке.

## Определение контрольного GUID и флагов

Каждый поставщик трассировки должен иметь контрольный GUID, с помощью которого контроллеры трассировки идентифицируют драйвер, как поставщика сообщений трассировки. Обычно, каждый драйвер имеет отдельный контрольный GUID. Но связь "один-к-одному" между контрольным GUID и драйвером не является обязательной по следующим причинам.

- ◆ Драйвер может иметь несколько контрольных GUID.

Каждый GUID должен идентифицировать уникального поставщика сообщений трассировки, но драйвер может быть разбит на несколько поставщиков, каждый с собственным контрольным GUID. Например, если драйвер имеет разделяемую библиотеку, он может определить два контрольных GUID: один собственно для драйвера, а другой для библиотеки. Но что касается флагов, то здесь каждый поставщик должен определять свой набор. Тогда потребитель трассировки может избрать каждого из них или обоих в качестве поставщика сообщений трассировки для сеанса.

- ◆ Несколько драйверов могут пользоваться одним контрольным GUID, и разные драйверы могут относиться к пользовательскому режиму и режиму ядра.

Все компоненты, пользующиеся одинаковым контрольным GUID, работают как один поставщик. Если поставщик разрешен, то сообщения трассировки от каждого компонента записываются в журнал вперемешку в едином порядке их поступления. Этот подход может быть полезен для работы с драйверами, взаимодействующими друг с другом при исполнении. Примером таких драйверов могут служить функциональный драйвер и соответствующий драйвер фильтра или гибридный драйвер, содержащий как UMDF-, так и KMDF-компоненты.

## Полезная информация

Пользуйтесь инструментом UuidGen, входящим в состав набора разработчика Platform SDK, для генерирования нового контрольного GUID специально для этой цели. Не копируйте примеров контрольных GUID из этой книги или используйте GUID, созданный для других целей, например, для идентификации интерфейса устройства. Дополнительную информацию см. в разделе **Using UUIDGEN.EXE** в библиотеке MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=79587>.

Флаги трассировки применяются для определения разных классов сообщений. Каждый драйвер может определить свой набор флагов трассировки с помощью макроса `WPP_DEFINE_CONTROL_GUID`. Определенная функция сообщения трассировки может генерировать сообщение, только если указанный флаг был включен контроллером трассировки. Чтобы указать контрольный GUID и флаги трассировки, необходимо определить макрос `WPP_CONTROL_GUIDS` для каждого исходного файла, который генерирует сообщения трассировки; обычно это делается в общем заголовочном файле. Каждому флагу необходимо присвоить имя и значение, указывающее бит, с которым ассоциирован данный флаг. Общая форма определения макроса показана в листинге 11.6.

#### Листинг 11.6. Формат определения макроса `WPP_CONTROL_GUIDS`

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(CtlGuid, \
        (aaaaaaaa, bbbb, cccc, dddd, eeeeeeeeeeee), \
        WPP_DEFINE_BIT(TRACELEVELONE) \
        WPP_DEFINE_BIT(TRACELEVELTWO) )
```

Значения компонентов определения следующие:

- ◆ `CtlGuid` — дружественный псевдоним для GUID;
- ◆ `(aaaaaaaa, bbbb, cccc, dddd, eeeeeeeeeeee)` — представляют сам GUID.
- Буквы представляют пять полей стандартной строковой формы GUID, но группы разделены запятыми вместо дефисов, а вся строка заключена в круглые, а не в фигурные скобки;
- ◆ `TRACELEVELONE` и `TRACELEVELTWO` — флаги трассировки.

WPP назначает битовое значение каждому флагу трассировки в порядке, в котором они следуют в определении макроса `WPP_CONTROL_GUIDS`, начиная с 1.

#### Пример UMDF: определение контрольного GUID и флагов

В образце драйвера `Fx2_Driver` используется один контрольный GUID и четыре флага трассировки, которые определены в заголовочном файле, как показано в листинге 11.7.

#### Листинг 11.7. Определение `WPP_DEFINE_CONTROL_GUID` для драйвера `Fx2_Driver`

```
#define WPP_CONTROL_GUIDS \
WPP_DEFINE_CONTROL_GUID(WudfOsrUsbFx2TraceCuid, \
    (da5fbdfd, leae, 4ecf, b426, a3818f325ddb), \
    WPP_DEFINE_BIT(MYDRIVER_ALL_INFO) \
    WPP_DEFINE_BIT(TEST_TRACE_DRIVER) \
    WPP_DEFINE_BIT(TEST_TRACE_DEVICE) \
    WPP_DEFINE_BIT(TEST_TRACE_QUEUE) \
)
```

Значения компонентов в данном примере следующие:

- ◆ `WudfOsrUsbFx2TraceCuid` — псевдоним для контрольного GUID `{da5fbdfd-leae-4ecf-b426-a3818f325ddb}`;

- ◆ элементы `WPP_DEFINE_BIT` задают четыре флага, используемые в параметре `FLAG` функции `TraceEvents`. Эти флаги применяются для различия между сообщениями, порождаемыми объектом драйвера, объектом устройства, объектами очереди и всеми прочими компонентами драйвера;
- ◆ параметр `LEVEL` функции автоматически соотносится со стандартными уровнями трассировки.

### Пример KMDF: определение `WPP_CONTROL_GUIDS` и флагов

В образце драйвера Osrusbf2 используется один контрольный GUID и семь флагов трассировки. Они определены в заголовочном файле Trace.h, как показано в листинге 11.8.

**Листинг 11.8. Определение `WPP_DEFINE_CONTROL_GUID` для драйвера Osrusbf2**

```
#define WPP_CONTROL_GUIDS \
WPP_DEFINE_CONTROL_GUID(OsrUsbFxTraceGuid, \
(d23a0c5a,d307,4f0e,ae8e,E2A355AD5DAB), \
WPP_DEFINE_BIT(DBG_INIT) /* bit 0 = 0x00000001 */ \
WPP_DEFINE_BIT(DBG_PNP) /* bit 1 = 0x00000002 */ \
WPP_DEFINE_BIT(DBG_POWER) /* bit 2 = 0x00000004 */ \
WPP_DEFINE_BIT(DBG_WMI) /* bit 3 = 0x00000008 */ \
WPP_DEFINE_BIT(DBG_CREATE_CLOSE) /* bit 4 = 0x00000010 */ \
WPP_DEFINE_BIT(DBG_IOCTL) /* bit 5 = 0x00000020 */ \
WPP_DEFINE_BIT(DBG_WRITE) /* bit 6 = 0x00000040 */ \
WPP_DEFINE_BIT(DBG_READ) /* bit 7 = 0x00000080 */ \
)
```

Значения компонентов в данном примере следующие:

- ◆ `OsrUsbFxTraceGuid` — псевдоним для контрольного GUID `{d23a0c5a-d307-4f0e-ae8e-E2A355AD5DAB}`;
- ◆ флаги трассировки применяются для различия между сообщениями трассировки, порождаемых при обработке разных типов запросов ввода/вывода.

## Инициализация и очистка трассировки

Трассировку необходимо инициализировать на раннем этапе процесса загрузки драйвера и деактивировать ее непосредственно перед остановкой драйвера.

### Инициализация трассировки

Трассировка инициализируется макросом `WPP_INIT_TRACING`, который необходимо вызывать на раннем этапе процесса загрузки, прежде чем генерировать какие-либо сообщения трассировки. Этот макрос определяется по-разному для драйверов режима ядра и драйверов пользовательского режима и принимает разные аргументы.

### Пример UMDF: инициализация трассировки в `DllMain`

Драйверы UMDF должны инициализировать трассировку в функции `DllMain` вызовом макроса `WPP_INIT_TRACING` при первом вызове функции. В данном случае параметру `Reason` функции присваивается значение `DLL_PROCESS_ATTACH`.

### Внимание!

Перед тем как включить трассировку, обязательно убедитесь в том, что параметру Reason присвоено значение DLL\_PROCESS\_ATTACH. В противном случае может возникать попытка включить трассировку при каждом подключении или отключении потока.

Для драйверов пользовательского режима макрос WPP\_INIT\_TRACING принимает один аргумент — указатель на строку в формате Unicode, идентифицирующую драйвер. Для образца драйвера Fx2\_Driver эта строка определена в заголовочном файле Internal.h, как показано в листинге 11.9.

#### Листинг 11.9. Определение константы MYDRIVER\_TRACING\_ID

```
#define MYDRIVER_TRACINC_ID L"Microsoft\\UMDF\\OsrUsb"
```

Применение константы позволяет копировать файл Dllsup.cpp и использовать его без изменений в другом драйвере. Но не нужно забыть переопределить константу MYDRIVER\_TRACING\_ID, назначив ей соответствующую строку для конкретного драйвера.

Константа MYDRIVER\_TRACING\_ID используется функцией DllMain при инициализации трассировки. В примере из листинга 11.10 приведена модифицированная версия реализации функции DllMain для образца драйвера Fx2\_Driver из файла Dllsup.cpp.

#### Листинг 11.10. Инициализация трассировки WPP в DllMain

```
BOOL WINAPI DllMain(HINSTANCE ModuleHandle,
                      DWORD Reason,
                      PVOID /* Reserved */)
{
    UNREFERENCED_PARAMETER(ModuleHandle);
    if (DLL_PROCESS_ATTACH == Reason)
    {
        WPP_INIT_TRACINC(MYDRIVER_TRACINC_ID);
    }
    . .
    return TRUE;
}
```

### Пример KMDF: инициализация трассировки в DriverEntry

Драйверы KMDF должны инициализировать трассировку в своей процедуре DriverEntry. Для драйверов режима ядра макрос WPP\_INIT\_TRACING принимает два аргумента: указатель на объект драйвера WDM и путь реестра. Оба аргумента передаются в качестве параметров функции DriverEntry.

### Примечание

В общем, процедуры для инициализации и очистки трассировки WPP для Windows 2000 отличаются от таковых для более поздних версий Windows. KMDF нейтрализует эту разницу в процедурах внутренней обработкой, поэтому следующие примеры можно применять на любой версии Windows, поддерживающей KMDF.

В примере из листинга 11.11 приводится несколько первых строчек процедуры DriverEntry образца драйвера Osrusbf2 из файла Device.c.

**Листинг 11.11. Инициализация трассировки WPP в DriverEntry**

```
NTSTATUS DriverEntry(_in PDRIVER_OBJECT DriverObject,
                     _in PUNICODE_STRING RegistryPath)
{
    WDF_DRIVER_CONFIG config;
    NTSTATUS status;
    WDF_OBJECT_ATTRIBUTES attributes;
    WPP_INIT_TRACING(DriverObject, RegistryPath);
    TraceEvents(TRACE_LEVEL_INFORMATION, DBG_INIT,
                "OSRUSBFX2 Driver Sample - Driver Framework Edition.\n");
    . .
}
```

**Очистка после трассировки**

Для деактивации трассировки драйверы должны вызвать макрос `WPP_CLEANUP`. После вызова этого макроса драйвер не может генерировать сообщения трассировки, поэтому этот вызов нужно делать непосредственно перед выгрузкой драйвера.

**Внимание!**

Если не вызвать макрос `WPP_CLEANUP`, то останется неосвобожденная ссылка на объект драйвера. Эта ссылка служит причиной, что последующие попытки загрузить драйвер завершаться неудачей, вследствие чего потребуется перезагрузка системы. А в случае попытки ETW обратиться к драйверу генерируется останов bugcheck. Имейте в виду, что упоминание вызвать макрос `WPP_CLEANUP` не вызывает ошибки компилятора при исполнении Build.exe.

**Пример UMDF: очистка после трассировки в DllMain**

Версия макроса `WPP_CLEANUP` для UMDF не принимает никаких аргументов. Этот макрос обычно вызывается в функции `DllMain` драйвера UMDF, которая вызывается как при загрузке, так и при выгрузке драйвера. При выгрузке драйвера параметру `Reason` присваивается значение `DLL_PROCESS_DETACH`. В примере из листинга 11.12 приводится модифицированная версия реализации функции `DllMain` для образца драйвера `Fx2_Driver` из файла `Dllsup.cpp`.

**Листинг 11.12. Очистка после трассировки WPP в функции DllMain**

```
BOOL WINAPI DllMain(HINSTANCE ModuleHandle,
                     DWORD Reason,
                     PVOID /* Reserved */)
{
    UNREFERENCED_PARAMETER(ModuleHandle);
    . .
    else if (DLL_PROCESS_DETACH == Reason)
    {
        WPP_CLEANUP();
    }
    return TRUE;
}
```

## Пример KMDF: очистка после трассировки WPP в функции *EvtCleanupCallback*

Драйвер KMDF должен зарегистрировать функцию обратного вызова *EvtCleanupCallback* для объекта типа *WDFDRIVER* в своей процедуре *DriverEntry* и вызывать макрос *WPP\_CLEANUP* из этой функции. Инфраструктура активирует функцию обратного вызова *EvtCleanupCallback* сразу же после удаления объекта драйвера и, таким образом, непосредственно перед выгрузкой драйвера.

Так как инфраструктура вызывает функцию *EvtCleanupCallback* только в случае успешного создания объекта драйвера, то WPP необходимо инициализировать только после успешного создания объекта драйвера. Таким образом, в случае неуспешного завершения попытки создания объекта драйвера, WPP не инициализируется, и выполнять очистку после нее не требуется. В случае же успешного создания объекта драйвера, инфраструктура гарантированно вызовет функцию *EvtCleanupCallback*, которая исполняет код очистки.

Версия макроса *WPP\_CLEANUP* для драйверов режима ядра принимает один аргумент: указатель на объект драйвера WDM. В примере из листинга 11.13 приводится реализация функции *EvtCleanupCallback* для образца драйвера *Osrusbf2* из файла *Device.c*.

### Листинг 11.13. Очистка после трассировки WPP в функции *EvtCleanupCallback*

```
VOID
OsrFxEvtDriverContextCleanup(IN WDFDRIVER Driver)
{
    PAGED_CODE ();
    TraceEvents(TRACE_LEVEL_INFORMATION, DBG_INIT,
                "<- OsrFxEvtDriverUnload\n");
    WPP_CLEANUP(WdfDriverWdmGetDriverObject(Driver));
}
```

Функции *EvtCleanupCallback* передается указатель на объект драйвера WDF. Чтобы получить указатель на лежащий в основе соответствующий объект драйвера WDM, образец драйвера передает дескриптор объекта WDF методу *WdfDriverWdmGetDriverObject*, который возвращает требуемый указатель.

### Полезная информация

Очистка после трассировки, выполняемая в функции *EvtCleanupCallback*, достаточна для большинства, но не для всех сценариев. Если попытка драйвера создать объект драйвера WDF в своей процедуре *DriverEntry* завершилась неудачей, то не существует объекта драйвера для удаления инфраструктурой, вследствие чего она не вызывает функцию *EvtCleanupCallback*. Чтобы обеспечить очистку после трассировки в этом случае, драйвер KMDF должен вызывать макрос *WPP\_CLEANUP* при обработке неудачной попытки создания объекта драйвера.

В примере из листинга 11.14 показана регистрация функции обратного вызова *EvtCleanupCallback* процедурой *DriverEntry* драйвера *Osrusbf2* и очистка трассировки в случае неудачного завершения попытки создания объекта драйвера.

### Листинг 11.14. Очистка трассировки WPP в процедуре *DriverEntry*

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

```

{
    ...
    attributes.EvtCleanupCallback = OsrfxEvtDriverContextCleanup;
    status = WdfDriverCreate(DriverObject,
                            RegistryPath,
                            &attributes,
                            &config,
                            &driver);
    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, DBG_INIT,
                    "WdfDriverCreate failed with status 0x%X\n",
                    status);
        WPP_CLEANUP(DriverObject);
        return status;
    }
}

```

## Оснащение кода драйвера

Вызовы функций сообщений трассировки в коде драйвера генерируют сообщения трассировки, получаемые потребителем трассировки. Эти функции можно разместить в любом требуемом месте кода, но обычно они вставляются в начале процедур или в ветвях ошибок.

### Пример UMDF: добавление вызовов функций сообщений трассировки в код драйвера

В примере из листинга 11.15 демонстрируется применение специальной функции трассировки, называющейся `Trace`, для возврата информации о состоянии ошибки. Данный код взят из файла Dllsup.cpp.

#### Листинг 11.15. Добавление сообщения трассировки в драйвер Fx2\_Driver

```

if (IsEqualCLSID(ClassId, CLSID_MyDriverCoClass) == false)
{
    Trace(TRACE_LEVEL_ERROR,
          ROR: Called to create instance of unrecognized class"(%!GUID!)",
          &ClassId);
    return CLASS_E_CLASSNOTAVAILABLE;
}

```

Эта функция `Trace` генерирует сообщения трассировки, только если потребитель включил флаг `TRACE_LEVEL_ERROR`.

### Пример KMDF: добавление вызовов функций сообщений трассировки в код драйвера

В следующем примере (листинг 11.16) демонстрируется применение функции `TraceEvents` драйвером Osrusbf2 в разделе кода, отвечающим за обработку запросов на чтение.

#### Листинг 11.16. Добавление сообщения трассировки в драйвер Osrusbf2

```

if (Length > TEST_BOARD_TRANSFER_BUFFER_SIZE) {
    TraceEvents(TRACE_LEVEL_ERROR, DBG_READ,
                "Transfer exceeds %d\n",
                TEST_BOARD_TRANSFER_BUFFER_SIZE);
}

```

```
Status = STATUS_INVALID_PARAMETER;  
}
```

Вызов функции `TraceEvents` генерирует сообщения трассировки, только если разрешен флаг `TRACE_LEVEL_ERROR` и установлен бит `DBG_READ`. Само сообщение содержит значение константы `TEST_BOARD_TRANSFER_BUFFER_SIZE`.

## Инструменты для программной трассировки

Инструментарий WDK содержит набор инструментов, как с графическим интерфейсом, так и с интерфейсом командной строки, для программной трассировки. Эти инструменты были разработаны для поддержки ETW и для расширения возможностей инструментов трассировки, предоставляемых Windows.

Инструменты трассировки WDK находятся по пути `%wdk%\tools\tracing\`. Для каждой поддерживаемой процессорной архитектуры имеется собственный набор. Далее приводится краткое описание этих инструментов.

### ◆ Logman.

Полнофункциональный контроллер трассировки с графическим интерфейсом, предназначенный для управления протоколированием счетчиков производительности и событий трассировки. Этот инструмент доступен, начиная с Windows XP.

### ◆ Tracefmt.

Потребитель трассировки с интерфейсом командной строки. Форматирует сообщения трассировки, поставляемые из сеансов трассировки режима реального времени или из журналов, и записывает их в файлы или выводит в окне командной строки.

### ◆ Tracelog.

Контроллер трассировки с интерфейсом командной строки. Поддерживает сеансы трассировки пользовательского режима и режима ядра.

### ◆ Tracepdb.

Инструмент командной строки для создания ТМН-файлов из файлов идентификаторов.

### ◆ Tracerpt.

Потребитель трассировки с интерфейсом командной строки. Форматирует события трассировки и счетчики производительности и записывает их файлы, читаемые другими приложениями, такими как, например, Microsoft Excel. Также анализирует события и генерирует сводные отчеты. Этот инструмент доступный, начиная с Windows XP.

### ◆ TraceView.

Контроллер и потребитель трассировки с графическим интерфейсом, сочетает и расширяет возможности инструментов Tracepdb, Tracelog и Tracefmt. Предназначен для вывода сообщений трассировки в режиме реального времени, но также может быть использован для просмотра файлов журналов трассировки.

### ◆ TraceWPP.

Инструмент командной строки, запускающий препроцессор WPP для обработки исходных файлов поставщика трассировки. Представляет альтернативу исполнению препроцессора в ходе обычного процесса сборки.

## Полезная информация

Журналы трассировки, созданные драйверами, использующими WPP, нельзя просматривать с помощью приложения Event Viewer, доступного из Панели управления (Control Panel).

# Проведение сеанса программной трассировки

Для проведения сеанса программной трассировки требуются следующие три компонента:

- ◆ контроллер трассировки, чтобы включить трассировки и соединить поставщика с потребителем;
- ◆ поставщик трассировки, чтобы генерировать сообщения трассировки;
- ◆ потребитель трассировки, чтобы форматировать сообщения и предоставлять их для просмотра.

Применяющийся в следующих примерах инструмент TraceView играет роль как контроллера, так и потребителя. В качестве потребителя трассировки и контроллера трассировки можно также использовать отдельные приложения. В этом разделе описывается использование инструмента TraceView с двумя образцами драйверов USB для подготовки драйвера к трассировке и последующего просмотра журналов трассировки.

## Подготовка драйвера

Прежде чем начать сеанс трассировки, необходимо скомпоновать и установить драйвер и создать TMF-файл и файл контрольного GUID (с расширением CTL). Процедура для драйвера Fx2\_Driver практически такая же, как и для драйвера Osrusbfx2, за исключением изменений некоторых имен.

### Чтобы подготовить драйвер:

1. Скомпонуйте и установите драйвер в тестовую систему.  
Описание инструкций по компоновке приводится в *главе 19*.
2. Скомпонуйте и установите тестовое приложение в тестовую систему.
3. Если на тестовой системе не установлен WDK, то скопируйте утилиту TraceView из системы разработки в удобное место на тестовой системе.
4. Из командной строки перейдите в целевой каталог (с помощью команды `cd`), после чего запустите утилиту Tracepdb, чтобы сгенерировать TMF-файлы из файла идентификаторов драйвера. В среде x86 для этого применяется следующая команда:

```
WinDDK\BuildNumber\tools\tracing\i386\tracepdb -f samplename.pdb -p path
```

Для 64-битных систем, вместо версии в папке `\i386`, необходимо использовать версию Tracepdb в папке `\x64` или `\ia64`.

Полученный TMF-файл сохраняется по пути `path`. Имя файла состоит из GUID с расширением TMF. Если команда исполняется из целевой папки, то вместо `path` используется `.` (точка).

### Примечание

Для Windows Server 2003 и более ранних версий необходимо скопировать файл `DbgHelp.dll` из папки `%wdk%\bin\Platform` в WDK в папку, в которой находится `Tracepdb.exe`.

## 5. Создайте для драйвера CTL-файл.

Это текстовый файл с одной строчкой текста, содержащей контрольный GUID и имя поставщика из макроса `WPP_DEFINE_CONTROL_GUID`. Файлу можно присвоить любое имя, но его расширение должно быть CTL.

Для образца драйвера `Fx2_Driver` содержимое CTL-файла должно быть следующим:

```
da5fbdfdf,1eae,4ecf,b426,a3818f325ddb WudfOsrUsbFx2TraceGuid
```

А для образца драйвера `Osrusbf2` содержимое CTL-файла должно быть таким:

```
d23a0c5a,d307,4f0e,ae8e,E2A355AD5DAB OsrUsbFxTraceGuid
```

## Просмотр журнала трассировки драйвера утилитой TraceView

Журналы трассировок можно просматривать в режиме реального времени или их можно сохранить в файл для просмотра позже. В этом разделе описывается процесс использования утилиты `TraceView` для просмотра журнала трассировки в режиме реального времени и для просмотра файла журнала трассировки.

В главе 22 приводятся инструкции по настройке отладчика `WinDbg` для просмотра журнала трассировки в режиме реального времени.

### Создание и просмотр файла журнала трассировки

В одном из способов обработки сообщений трассировки они сохраняются в файл журнала трассировки, который можно просмотреть с помощью потребителя трассировки позже. В этом разделе показано, как сохранять сообщения трассировки из драйвера `Fx2_Driver` в файл журнала трассировок, а потом просматривать их с помощью утилиты `TraceView`.

#### Чтобы начать сеанс трассировки:

1. Установите драйвер `Fx2_Driver` и утилиту `TraceView` на компьютер, используемый для тестирования.
2. Запустите утилиту `TraceView`, дважды щелкнув по ее значку в Проводнике Windows (Windows Explorer) или из командной строки.

В `Window Vista` и более поздних версиях для запуска `TraceView` необходимо иметь повышенные привилегии.

3. В меню **File** щелкните мышью элемент **Create New Log Session**, после чего выберите элемент **Add Provider**.
4. В диалоговом окне **Provider Control GUID Setup** выберите **CTL (Control GUID) File** и введите путь к CTL-файлу образца драйвера, после чего щелкните кнопку **OK**.
5. В диалоговом окне **Format Information Source Select** выберите опцию **Select TMF Files** и нажмите кнопку **OK**.
6. В диалоговом окне **Trace Format Information Setup** нажмите кнопку **Add**.
7. В диалоговом окне **File Open** введите созданные ранее TMF-файлы.
8. В диалоговом окне **Trace Format Information Setup** нажмите кнопку **Done**.
9. В диалоговом окне **Create New Log Session** нажмите кнопку **Next**.

10. На странице **Log Session Options** выберите опцию **Log Trace Event Data To File**, укажите имя для файла журнала, после чего завершите подготовку сеанса трассировки, нажав кнопку **Finish**.

Драйвер Fx2\_Driver имеет сообщения трассировки только в своем коде загрузки и запуска. Чтобы создать журнал сообщений трассировки, отключите устройство из разъема и вставьте его обратно в разъем.

#### Чтобы просмотреть сообщения в файле журнала:

1. Запустите утилиту TraceView.
2. В меню **File** щелкните элемент **Open Existing Log File**.
3. В диалоговом окне **Log File Selection** введите имя созданного в предыдущей процедуре файла журнала трассировки и нажмите кнопку **OK**.
4. В диалоговом окне **Format Information Setup** укажите TMF-файлы проекта, после чего нажмите кнопку **Done**, чтобы начать просмотр журнала сообщений трассировки.

На рис. 11.2 показано содержимое файла журнала трассировок для драйвера Fx2\_Driver, полученное в результате отключения устройства из разъема и обратного подключения при исполняющемся сеансе.

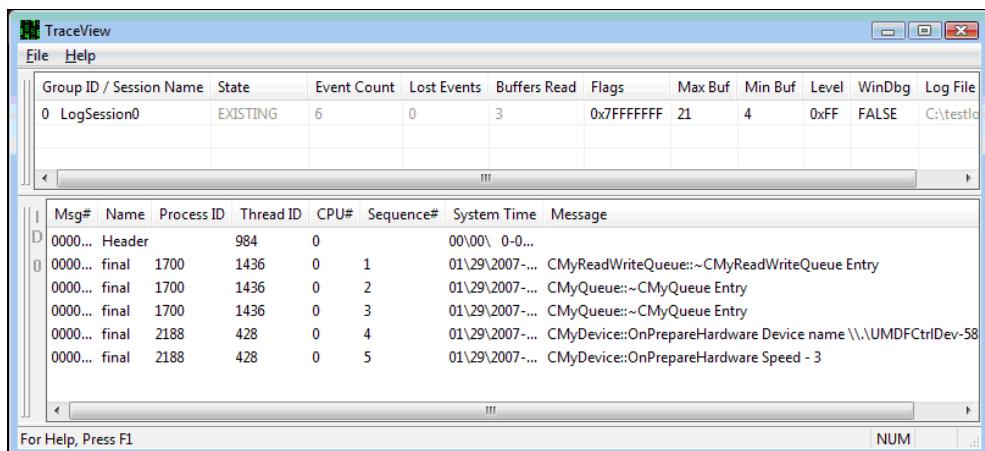


Рис. 11.2. Просмотр журнала трассировки драйвера утилитой TraceView

#### Просмотр журнала трассировки в режиме реального времени

Просматривать журнал трассировок в режиме реального времени часто удобно, чтобы увидеть, каким образом драйвер реагирует на различные события. В этом разделе описывается простой сеанс трассировки с образцом драйвера Osrusbf2.

#### Чтобы начать сеанс трассировки:

1. Установите драйвер Osrusbf2, тестовое приложение и утилиту на тестовый компьютер.
2. Запустите утилиту TraceView.
3. В меню **File** щелкните мышью элемент **Create New Log Session**, после чего выберите элемент **Add Provider**.

4. В диалоговом окне **Provider Control GUID Setup** выберите **CTL (Control GUID) File** и введите путь к CTL-файлу образца драйвера, после чего щелкните кнопку **OK**.
5. В диалоговом окне **Format Information Source Select** выберите опцию **Select TMF Files** и нажмите кнопку **OK**.
6. В диалоговом окне **Trace Format Information Setup** нажмите кнопку **Add**.
7. В диалоговом окне **File Open** введите созданные ранее TMF-файлы, после чего нажмите кнопку **Done**.
8. В диалоговом окне **Create New Log Session** нажмите кнопку **Next**.
9. На странице **Log Session Options** выберите опцию **Real Time Display**, после чего завершите подготовку сеанса трассировки, нажав кнопку **Finish**.

В добавление к просмотру сообщений в режиме реального времени их можно также сохранять в файл журнала трассировок.

**Чтобы генерировать сообщения трассировки**, запустите тестовое приложение и внесите изменение в какой-либо аспект функционирования устройства.

На рис. 11.3 показаны сообщения трассировки, полученные в результате включения двух светодиодов на светодиодной линейке устройства обучения OSR USB FX2.

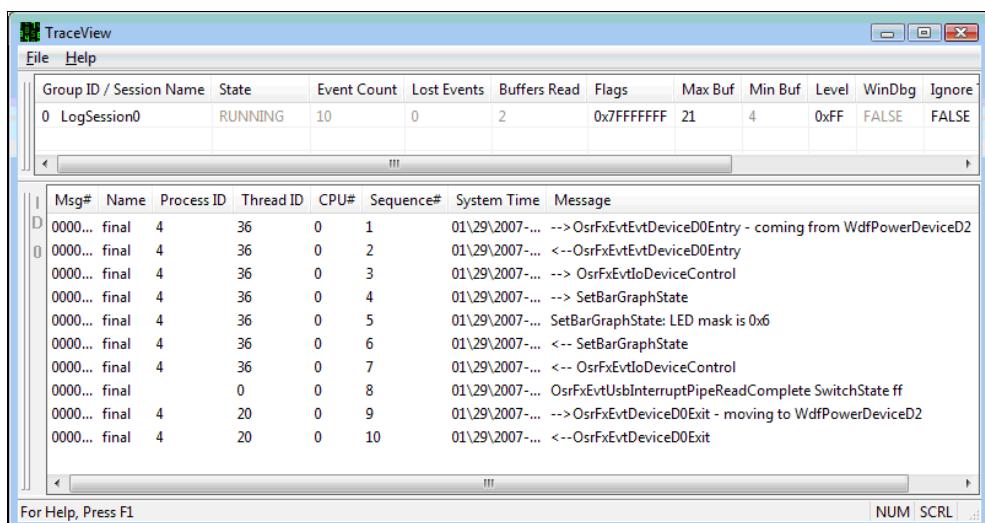


Рис. 11.3. Просмотр сообщений трассировки в режиме реального времени

## Просмотр инфраструктурного журнала трассировки с помощью базовых инструментов трассировки

Базовые утилиты трассировки Tracelog и Tracefmt менее удобны в пользовании, чем утилита TraceView. Но с другой стороны, они более надежные, что делает их применение предпочтительным в некоторых ситуациях. В этом разделе показано, как использовать базовые инструменты трассировки для просмотра сообщений трассировки из среды исполнения KMDF.

Чтобы просмотреть сообщения из среды исполнения с помощью утилиты Tracelog, откройте окно консоли и выполните команды, приведенные в листинге 11.17, чтобы установить несколько переменных среды и создать папку для файла журнала трассировки.

#### Листинг 11.17. Установка переменных среды с помощью утилиты Tracelog

```
set TRACE_NAME = WDF
set TRACE_LOG_PATH = %SystemRoot%\Tracing\%TRACE_NAME%
set _PROVIDER_GUID = #544d4c9d-942c-46d5-bf50-df5cd9524a50
set TMFFILE = wdf01005.tmf
set TRACE_FORMAT_PREFIX=%%2!-20.20s!%%! FUNC! -
set TRACE_FLAGS = 0xffff
set TRACE_LEVEL = 5
md "%TRACE_LOG_PATH%"
```

Фрагменты кода в листинге 11.17, выделенные жирным шрифтом, необходимо модифицировать значениями, подходящими для конкретного драйвера, в особенности значения GUID и имя TMF-файла. При исполнении утилиты Tracelog на системах под управлением Windows Vista окно консоли необходимо открывать с учетной записи с повышенными привилегиями.

Чтобы начать сеанс Tracelog, выполните следующую команду:

```
tracelog -start "%TRACE_NAME%" -seq 10 -rt -guid %_PROVIDER_GUID% -flags %TRACE_FLAGS%
-level %TRACE_LEVEL% -f "%TRACE_LOG_PATH%\%TRACE_NAME%.etl"
```

По умолчанию сообщения помещаются в файл журнала трассировки с именем %TRACE\_NAME%, который находится в папке %TRACE\_LOG\_PATH%. Сообщения также можно просматривать в режиме реального времени. Для этого после начала сеанса Tracelog запустите утилиту Tracefmt.

Чтобы просматривать сообщения в режиме реального времени утилитой Tracefmt, выполните следующую команду:

```
tracefmt -rt "%TRACE_NAME%" -tmf "%TMFFILE%" -display
```

По мере создания сообщений утилита Tracefmt выводит их в окно командной строки, форматируя их согласно информации в указанном TMF-файле. В листинге 11.18 приведен фрагмент журнала сообщений, выводимый утилитами Tracelog и Tracefmt в режиме реального времени. Сообщения были порождены включением двух светодиодов на светодиодной линейке устройства обучения OSR USB FX2.

#### Листинг 11.18. Вывод сообщений в режиме реального времени утилитами Tracelog и Tracefmt

```
C:\WinDDK\6000\tools\tracing\i386>tracefmt.exe -rt "%TRACE_NAME%"
                                              -tmf "XTMFFILE"
%" -display
Setting Realtime mode for WDF
Examining wdf01005.tmf for message formats, 97 found.
Searching for TMF files on path: (null)
FxPkgGeneral_cpp424  FxPkgGeneral::Dispatch- WDFDEVICE 0x6C3BC458
  !devobj 0x93C44810 0x00000000(IRP_MJ_CREATE)  IRP 0x9061A008
FxPkgIo_cpp97        FxPkgIo::Dispatch- WDFDEVICE 0x6C3BC458
  !devobj 0x93C44810 0x0000000e(IRP_MJ_DEVICE_CONTROL),    IRP_MN 0,
  IRP 0x9331FED8
```

```

FxDevice_cpp1646      FxDevice::AllocateRequestMemory- Allocating
    FxRequest* 9048E308, WDFREQUEST 6FB71CFO
FxIoQueue_cpp1696      FxIoQueue::QueueRequest- Queuing WDFREQUEST
    0x6FB71CFO on WDFQUEUE 0x6C3BAFB0
FxIoQueue_cpp2085      FxIoQueue::DispatchEvents- Thread 9E867AC0
    is processing WDFQUEUE 0x6C3BAFB0
PowerPolicyStateMachFxPkgPnp::PowerPolicyCancelWaitWake-
    Successfully got WaitWake irp 93C49750 for canceling
PowerPolicyStateMachFxPkgPnp::_PowerPolicyWaitWakeCompletionRoutine-
    Completion of WaitWake irp 93C49750, 0xc0000120(STATUS_CANCELLED)
PowerPolicyStateMachFxPkgPnp::_PowerPolicyWaitWakeCompletionRoutine-
    Not completing WaitWake irp 93C49750 in completion routine
PowerPolicyStateMachFxPkgPnp::PowerPolicyCancelWaitWake- Cancel of
    irp 93C49750 returned 1
FxPkgPnp_cpp494 FxPkgPnp::Dispatch- WDFDEVICE 0x6C3BC458
    !devobj 0x93C44810 IRP_MJ_POWER, Minor 0x2 IRP 0x93C49750
PowerPolicyStateMachFxPkgPnp::PowerPolicySendDevicePowerRequest-
    Requesting DO irp, 0x00000103(STATUS_PENDING)
FxIoTarget_cpp158      FxIoTarget::SubmitPendedRequest- Sending
    WDFREQUEST 6CE5CCC8, Irp 931A52F8
FxIoTarget_cpp158      FxIoTarget::SubmitPendedRequest- Sending
    WDFREQUEST 6C3BB268, Irp 93C49B38
FxIoQueue_cpp2085      FxIoQueue::DispatchEvents- Thread 820FBD78
    is processing WDFQUEUE 0x6C3BAFB0
FxIoQueue_cpp2170      FxIoQueue::DispatchEvents- WDFQUEUE
    0x6C3BAFB0 Power Transition State
    0x0000000a(FxIoQueuePowerRestarting)

```

Чтобы сбросить буфер трассировки и просмотреть сообщения, не останавливая сеанса:

1. Чтобы сбросить буфер, выполните следующую команду:

```
tracelog -flush "%TRACE_NAME%"
```

2. Потом выполните следующую команду:

```
tracefmt "%TRACE_LOG_PATH%\%TRACE_NAME%.etl" -tmf %TMFFILE%-
nosummary -o "%TRACE_LOG_PATH%\%COMPUTERNAME%-
%TRACE_NAME%.txt"
```

3. Наконец, чтобы поместить сообщения в файл, выполните следующую команду:

```
@echo Tracelog dumped to %TRACE_LOG_PATH%\%COMPUTERNAME%-
%TRACE_NAME%.txt
```

Чтобы остановить сеанс трассировки, выполните следующую команду:

```
tracelog -stop "%TRACE_NAME%"
```

## **Практические рекомендации: думайте о диагностике**

Генерирование сообщений трассировки предоставляет относительно легкий способ получить и сохранить информацию о работе драйвера. С помощью сообщений трассировки можно отследить путь исполнения запроса ввода/вывода в коде, вычислить местонахожде-

ние ошибок и определить, какие процедуры вызываются часто, а какие редко. Наличие такого рода информации позволяет облегчить диагностику проблем в драйвере.

Чтобы получить наибольшую отдачу от трассировки, следуйте этим практическим рекомендациям.

- ◆ Определяйте флаги трассировки, чтобы они соответствовали типам информации, которую вы считаете наиболее полезной при отладке.

Например, при инкрементной реализации и тестировании кода можно определять новый флаг для каждого нового модуля драйвера.

- ◆ Не подвергайте операторы трассировки в драйвере условной компиляции.

Оставьте эти операторы в двоичных файлах, поставляемых клиенту. Сообщения трассировки оказывают заметное влияние на производительность, только если пользователь явно включит их.

- ◆ Вставляйте сообщение трассировки в каждую ветвь исполнения кода ошибки.

Такие сообщения могут предоставить подробную информацию и помочь в диагностировании ошибки со сравнительно небольшими накладными расходами.

# ГЛАВА 12

## Вспомогательные объекты WDF

Кроме объектов драйвера, устройства, очереди и запроса ввода/вывода, представляющих основные разделяемые всеми драйверами абстракции, WDF также содержит несколько других типов объектов, поддерживающих возможности, которые могут время от времени требоваться драйверам. В этой главе описывается применение таких объектов. Также в ней рассматривается поддержка для инструментария WMI,строенная в KMDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b> Pooltag.txt	%wdk%\tools\other\platform\poolmon
<b>Образцы драйверов</b> Toastmon Featured Toaster	%wdk%\Src\Kmdf\Toaster\Toastmon\ %wdk%\Src\Kmdf\Toaster\Func\Featured
<b>Документация WDK</b> Раздел "ExAllocatePoolWithTag" Introduction to WMI <sup>1</sup> Kernel-Mode Driver Framework Design Guide <sup>2</sup> User-Mode Driver Framework Design Guide <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82323">http://go.microsoft.com/fwlink/?LinkId=82323</a> <a href="http://go.microsoft.com/fwlink/?LinkId=82322">http://go.microsoft.com/fwlink/?LinkId=82322</a> <a href="http://go.microsoft.com/fwlink/?LinkId=79342">http://go.microsoft.com/fwlink/?LinkId=79342</a> <a href="http://go.microsoft.com/fwlink/?LinkId=79341">http://go.microsoft.com/fwlink/?LinkId=79341</a>

## Выделение памяти

Драйверы WDF используют и выделяют память в качестве общего ресурса несколькими специфичными способами:

- ◆ под локальное хранилище;
- ◆ как объекты памяти WDF.

<sup>1</sup> Введение в инструментарий WMI. — Пер.

<sup>2</sup> Руководство по разработке в инфраструктуре для написания драйверов режима ядра (KMDF). — Пер.

<sup>3</sup> Руководство по разработке в инфраструктуре для написания драйверов пользовательского режима (UMDF). — Пер.

В этом разделе предоставляется основная информация о выделении памяти под локальное хранилище и о создании и выделении буферов и объектов памяти.

## Локальное хранилище

В некоторых ситуациях драйверу требуется локальное хранилище, которое не ассоциировано с запросом ввода/вывода или которое нужно передать за пределы инфраструктуры другому компоненту.

В зависимости от типа требуемого хранилища, драйверы UMDF могут применять для выделения памяти такие методы, как `new` и `malloc`, а также другие методы Windows пользовательского режима и методы, ориентированные на конкретный язык программирования.

Драйверы KMDF применяют интерфейсы DDI режима ядра Windows (обычно процедуру `ExAllocatePoolWithTag`) для выделения памяти, не являющейся частью объекта памяти WDF. Например, процедуру `ExAllocatePoolWithTag` используют следующие образцы драйверов.

- ◆ Образец драйвера Firefly выделяет буфер для отправки вниз по своему стеку устройств в синхронном запросе IOCTL.

В синхронном запросе использование объекта памяти WDF вызывает дополнительные накладные расходы, не принося никакой пользы. Так как драйвер ожидает завершения запроса, то маловероятно, что память будет освобождена в несоответствующее время.

- ◆ Образец драйвера Featured Toaster выделяет память для использования в вызовах функций семейства `IoWmiXxx`.

В данном случае функции семейства `IoWmiXxx` освобождают память от имени вызывающего клиента. Драйверы WDF не должны использовать объекты памяти WDF в вызовах функций семейства `IoWmiXxx`, т. к. такие функции не могут освобождать объекты памяти.

- ◆ Образец драйвера KMDF 1394 выделяет память для блоков управления, которые он отправляет своему драйверу шины.

Драйвер шины ожидает параметры, упакованные в блоке запроса ввода/вывода IEEE 1394.

Процедура `ExAllocatePoolWithTag` может выделять память или из страничного, или нестраничного пула. Инфраструктура не отслеживает ссылки на память, выделяемую системными функциями, поэтому перед выгрузкой драйвер должен вызывать процедуру `ExFreePoolWithTag`, чтобы освободить память, выделенную процедурой `ExAllocatePoolWithTag`.

В файле Pooltag.txt перечислены теги пулов, которые используются компонентами режима ядра и драйверами, поставляемые с Windows вместе с ассоциированным файлом или компонентом, и имя компонента. Файл Pooltag.txt устанавливается с инструментарием Debugging Tools for Windows (в папке %windbg%\triage) и с набором разработчика WDK (в папке %wdk%\tools\other\platform\poolmon, где *platform* означает amd64, i386 или ia64).

Общую информацию по процедуре `ExAllocatePoolWithTag` и другим процедурам WDM для выделения памяти см. в разделе **Allocating System-Space Memory** (Выделение памяти пространства системного ядра) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81580>.

## Объекты памяти и буферы ввода/вывода

Объект памяти WDF — это объект с подсчитываемыми ссылками, который описывает буфер. Запросы ввода/вывода, передаваемые инфраструктурой драйверу, содержат объекты памяти WDF, описывающие буферы, в которых драйвер получает и возвращает данные.

Применение объектов памяти WDF не ограничено использованием их в запросах ввода/вывода. Драйвер также может использовать объекты памяти WDF внутренне для других целей, например для внутренних буферов, используемых совместно несколькими функциями драйвера. Отслеживание числа ссылок уменьшает шансы, что драйвер ненароком освободит память в неподходящий момент или попытается обратиться к памяти, которая уже была освобождена. По умолчанию родителем созданного драйвером объекта памяти является сам драйвер, поэтому по умолчанию объект памяти продолжает существовать до тех пор, пока драйвер не выгрузится.

Каждый объект памяти содержит размер буфера, который он представляет. Методы WDF, которые копируют данные в буфер и из буфера, проверяют размер переданных данных при каждой операции передачи, с тем, чтобы предотвратить переполнение или опустошение буфера, в результате чего данные могут быть искажены или может возникнуть угроза безопасности.

Дополнительная информация об использовании объектов памяти и буферов в запросах ввода/вывода и их времени жизни приводится в *главах 8 и 9*.

## Объекты памяти UMDF и их интерфейсы

Независимо от того, каким образом драйвер использует объект памяти, для создания объекта памяти и ассоциированного с ним буфера драйверы UMDF применяют те же самые методы. Обычно драйвер выделяет буфер и создает ассоциированный объект памяти одновременно, вызывая метод `IWDFDriver::CreateWdfMemory`, поэтому время жизни объекта памяти и буфера одинаковые. Если для буфера требуется более продолжительное время жизни, чем для объекта памяти, драйвер должен использовать метод `IWDFDriver::CreatePreallocatedWdfMemory`.

Дополнительные сценарии создания буферов и объектов памяти и задания их времени жизни приводятся в *главе 9*.

Драйвер манипулирует объектом памяти и обращается к ассоциированному буферу посредством методов интерфейса `IWDFMemory`. Краткое описание этих методов приводится в табл. 12.1.

**Таблица 12.1. Методы `IWDFMemory`**

Метод	Описание
<code>CopyFromBuffer</code>	Копирует данные из буфера источника в объект памяти
<code>CopyFromMemory</code>	Копирует данные из одного объекта памяти в другой объект памяти и предотвращает переполнение, которое в противном случае могла бы вызвать операция копирования
<code>CopyToBuffer</code>	Копирует данные из объекта памяти в буфер
<code>GetDataBuffer</code>	Извлекает буфер данных, ассоциированный с объектом памяти
<code>GetSize</code>	Извлекает размер буфера данных, ассоциированного с объектом памяти
<code>SetBuffer</code>	Назначает буфер объекту памяти, созданному драйвером вызовом метода <code>IWDFDriver::CreatePreallocatedWdfMemory</code>

Интерфейс `IWDFIoRequest` содержит методы, с помощью которых драйвер может извлекать объекты памяти, вложенные в запросы ввода/вывода.

Дополнительная информация об этих методах предоставляется в *главе 8*.

## Объекты памяти KMDF и их методы

Драйвер KMDF может одновременно выделить буфер и создать объект памяти, вызывая метод `WdfMemoryCreate`. Если драйвер часто использует буферы одинакового размера, то он может создать список резервных буферов (lookaside list), содержащий буферы требуемого типа, после чего назначать буферы из этого списка новым объектам памяти, вызывая метод `WdfMemoryCreateFromLookaside`.

Для манипулирования объектами памяти WDF и для чтения и записывания их буферов инфраструктура определяет методы семейства `WdfMemoryXxx`. Эти методы принимают дескриптор объекта памяти в качестве параметра и перемещают данные между буфером объекта памяти и внешним буфером. Краткое описание этих методов приводится в табл. 12.2.

**Таблица 12.2. Объекты памяти KMDF и их методы**

Метод	Описание
<code>WdfMemoryAssignBuffer</code>	Назначает указанный буфер объекту памяти, созданному драйвером
<code>WdfMemoryCopyFromBuffer</code>	Копирует данные из исходного буфера в буфер объекта памяти
<code>WdfMemoryCopyToBuffer</code>	Копирует данные из буфера объекта памяти в другой буфер
<code>WdfMemoryCreate</code>	Создает объект типа <code>WDFMEMORY</code> и выделяет буфер памяти указанного размера
<code>WdfMemoryCreateFromLookaside</code>	Создает объект типа <code>WDFMEMORY</code> и назначает ему буфер со списка резервных буферов
<code>WdfMemoryCreatePreallocated</code>	Создает объект типа <code>WDFMEMORY</code> и назначает ему существующий буфер, предоставленный драйвером
<code>WdfMemoryGetBuffer</code>	Возвращает указатель на буфер, ассоциированный с объектом памяти

Отправляемый инфраструктурой драйверу KMDF запрос ввода/вывода содержит один или несколько объектов типа `WDFMEMORY`. Драйверы могут обращаться к объектам памяти и буферам в запросе ввода/вывода с помощью методов семейства `WdfRequestRetrieveXxx`.

## Списки резервных буферов KMDF

Список резервных буферов представляет собой список буферов фиксированного размера многократного использования, предназначенных для структур, выделяемых драйвером динамически на регулярной основе. Список резервных буферов полезен в любой ситуации, когда драйверу требуются буферы фиксированного размера, и особенно подходит для широко применяемых и повторно применяемых структур. Например, менеджер ввода/вывода Windows выделяет свои пакеты IRP со списка резервных буферов.

Драйвер определяет размер буферов, а система содержит статус списка и регулирует число имеющихся в наличии буферов соответственно спросу. В зависимости от требований драйвера, список резервных буферов можно выделить либо из страничного, либо из нестраничного пула. После инициализации списка все буферы в списке выделяются из одного и того же пула.

Когда драйвер инициализирует список резервных буферов, Windows создает список и держит буферы в резерве для использования драйвером в будущем. Текущее число буферов

в списке зависит от объема доступной памяти и размера буферов. Когда драйверу требуется буфер, он вызывает инфраструктуру, чтобы создать объект памяти из списка резервных буферов. Объект памяти инкапсулирует буфер и устанавливается владельцем буфера, так что время жизни буфера такое же, как и времени жизни объекта памяти. Когда объект памяти удаляется, соответствующий буфер возвращается в список резервных буферов.

### Пример: использование списка резервных буферов

Драйвер KMDF создает список резервных буферов с помощью метода `WdfLookasideListCreate`. По умолчанию родителем списка резервных буферов является объект драйвера. Чтобы указать иного родителя, драйвер устанавливает поле `ParentObject` в структуре атрибутов объекта списка. Когда инфраструктура удаляет родительский объект, она также удаляет и список. Драйвер также может удалить список самостоятельно, вызывая метод `WdfObjectDelete`.

В листинге 12.1 приведен пример создания образцом драйвера `Pcidrv` списка резервных буферов с областью видимости драйвера. (Данный фрагмент исходного кода взят из файла `Sys\Pcidrv.c`.)

#### Листинг 12.1. Создание списка резервных буферов

```
status = WdfLookasideListCreate(WDF_NO_OBJECT_ATTRIBUTES,
                                sizeof(MP_RFD),
                                NonPagedPool,
                                WDF_NO_OBJECT_ATTRIBUTES, // MemoryAttributes
                                PCIDRV_POOL_TAG,
                                &driverContext->RecvLookaside);
```

Образец драйвера `Pcidrv` использует список резервных буферов в качестве источника буферов для принимающих структур. Один список резервных буферов с областью видимости драйвера обслуживает все устройства, управляемые драйвером, поэтому драйвер создает список в своей функции `DriverEntry` и сохраняет дескриптор списка, возвращенного функцией, в области контекста драйвера. В данном примере список выделяется из нестраничного пула.

Каждый буфер в списке имеет размер в байтах `sizeof(MP_RFD)` и использует тег пула, определенный в константе `PCIDRV_POOL_TAG`. Использование уникального тега пула для драйвера, или даже для индивидуальных модулей драйвера, является важным аспектом, т. к. тег помогает определить источник памяти при множественных выделениях памяти при отладке. Метод `WdfLookasideListCreate` принимает в качестве первых двух параметров указатели на две структуры атрибутов объектов. Первая структура атрибутов описывает атрибуты самого объекта списка резервных буферов, а вторая структура атрибутов описывает атрибуты объектов `WDFMEMORY`, которые драйвер впоследствии выделяет из списка. Драйвер в листинге 12.1 не указывает никаких атрибутов ни для каких типов объектов.

Драйвер выделяет буфер из списка, вызывая метод `WdfMemoryCreateFromLookaside`. Этот метод возвращает объект типа `WDFMEMORY`, который драйвер может использовать, как любой другой объект `WDFMEMORY`.

В листинге 12.2 образец драйвера `PCIDRV` выделяет объект памяти со своего списка резервных буферов, после чего получает указатель на буфер в этом объекте. Исходный код этого примера адаптирован из файла `Pcidrv\sys\hw\Nic_init.c`.

**Листинг 12.2. Выделение памяти из списка резервных буферов**

```

NTSTATUS NICInitRecvBuffers(IN PFDO_DATA FdoData)
{
    NTSTATUS status = STATUS_INSUFFICIENT_RESOURCES;
    PMP_RFD pMpRfd;
    WDFMEMORY memoryHdl;
    PDRIVER_CONTEXT driverContext = GetDriverContext(WdfGetDriver());

    . . . // Код опущен, чтобы сохранить место.
    status = WdfMemoryCreateFromLookaside(driverContext->RecvLookaside,
                                           &memoryHdl);

    if (!NT_SUCCESS(status)){
        . . . // Код для обработки ошибки опущен.
    }

    pMpRfd = WdfMemoryGetBuffer(memoryHdl, NULL);
    if (IpMpRfd) {
        . . . // Код для обработки ошибки опущен.
    }
    . . . // Нерелевантный код опущен.
    return;
}

```

В представленном листинге образец драйвера передает дескриптор объекта списка резервных буферов, который он ранее сохранил в области контекста драйвера, методу `WdfMemoryCreateFromLookaside`, который возвращает дескриптор объекта типа `WDFMEMORY` в переменной `memoryHdl`. После проверки статуса драйвер вызывает метод `WdfMemoryGetBuffer`, который возвращает указатель на буфер, вложенный в объект памяти. В первом параметре методу передается дескриптор объекта памяти, а во втором — адрес, по которому нужно возвратить размер буфера. В данном случае во втором параметре драйвер передает значение `NULL`, т. к. он уже указал размер буфера при создании списка резервных буферов.

Когда драйвер больше не нуждается в объекте памяти, он должен его удалить с помощью метода `WdfObjectDelete` следующим образом:

```
WdfObjectDelete(memoryHdl);
```

Когда драйвер удаляет объект памяти, инфраструктура возвращает соответствующий буфер в список резервных буферов.

## Обращение к реестру

Драйверы могут использовать реестр для получения информации о своих устройствах и для хранения информации, которую необходимо сохранить между перезагрузками операционной системы. UMDF и KMDF предоставляет для драйверов следующие способы для чтения и записи в реестр:

- ◆ драйверы UMDF используют хранилище свойств устройства;
- ◆ драйверы KMDF применяют объект раздела реестра.

## Хранилище свойств устройства UMDF

Хранилище свойств устройства представляет собой область реестра, в которой драйвер UMDF может хранить информацию о характеристиках устройства. В качестве примера та-

ких характеристик можно назвать значения тайм-аута или конфигурационные установки, т. е. любая специфическая для устройства информация, которую драйвер сохраняет, чтобы применять при каждом запуске системы или устройства.

Каждое хранилище свойств имеет имя, совпадающее с именем раздела реестра, в котором хранится информация. По умолчанию имя раздела такое же, как и имя драйвера. Хранилище свойств содержит один или несколько именованных строковых, целочисленных или двоичных параметров. Для чтения информации из хранилища свойств драйвер UMDF должен иметь как имя хранилища данных, так и имя параметра, содержащего данные.

Хранилище свойств предоставляет безопасный способ для драйверов UMDF записывать данные в реестр и изолирует драйверы от настоящего месторасположения данных. Драйверы UMDF не должны пытаться читать или записывать данные в хранилище свойств, обращаясь к определенной ячейке реестра, и наоборот, драйверы UMDF не могут использовать методы хранилища свойств для чтения или записи данных, таких как параметры устройства, которые записываются в других разделах реестра. Чтобы получить информацию из какого-либо другого раздела реестра, драйверы должны пользоваться интерфейсом Registry API или Setup API. Но драйверы UMDF исполняются в контексте безопасности LocalService, который ограничивает число областей реестра, к которым драйвер может иметь доступ для чтения и записи.

Драйверы UMDF могут создать новое хранилище свойств устройства или получить доступ к существующему хранилищу свойств, вызывая метод `RetrieveDevicePropertyStore` любого из следующих интерфейсов:

- ◆ `IWDFDeviceInitialize` — во время инициализации;
- ◆ `IWDFDevice` — после создания объекта устройства.

Метод `RetrieveDevicePropertyStore` возвращает указатель на вспомогательный интерфейс `IWDFNamedPropertyStore`, через который драйвер может устанавливать и получать значение свойств устройства. Этот метод принимает следующие четыре параметра:

- ◆ `pcwszServiceName` — указатель на строку с завершающим нулем, представляющую имя хранилища свойств устройства, или значение `NULL`, чтобы использовать имя вызывающего драйвера в списке сервисов WUDF;
- ◆ `Flags` — значение перечисления `WDF_PROPERTY_STORE_FLAGS`:
  - если хранилище свойств не существует, то при указании значения `WdfPropertyStoreNormal` оно не создается;
  - если хранилище свойств не существует, то при указании значения `WdfPropertyStoreCreateIfMissing` оно создается;
- ◆ `ppPropStore` — указатель на буфер, в который метод возвращает указатель на интерфейс `IWDFNamedPropertyStore`;
- ◆ `pDisposition` — указатель на переменную, в которую метод возвращает одно из следующих значений перечисления `WDF_PROPERTY_STORE_DISPOSITION`:
  - значение `CreatedNewStore` означает, что инфраструктура создала новое хранилище свойств;
  - значение `OpenedExistingStore` означает, что инфраструктура открыла существующее хранилище свойств.

После создания или получения драйвером хранилища свойств, он устанавливает или получает значения из него посредством интерфейса `IWDFNamedPropertyStore`. Сводка методов этого интерфейса приводится в табл. 12.3.

**Таблица 12.3. Методы интерфейса `IWDFNamedPropertyStore`**

Метод	Описание
<code>GetNameAt</code>	Получает имя свойства по его индексу
<code>GetNameCount</code>	Извлекает число свойств, содержащихся в хранилище данных
<code>GetNamedValue</code>	Получает значение свойства по имени параметра
<code>SetNamedValue</code>	Устанавливает значение свойства

Формат и содержимое хранилища свойств определяются драйвером. Хранилище свойств продолжает существовать в реестре до тех пор, пока устройство не будет deinсталлировано из системы. Инфраструктура содержит хранилище свойств в узле `devnode`, так что оно удаляется при удалении устройства. Поставщик устройства не обязан предоставлять процедуру для deinсталляции. В листинге 12.3 приводится пример создания драйвером именованного хранилища свойств после создания им объекта устройства. Исходный код этого примера взят из файла `Sideshow\WSSDevice.cpp`.

#### Листинг 12.3. Создание именованного хранилища свойств

```
hr = m_pWdfDevice->RetrieveDevicePropertyStore(NULL,
    WdfPropertyStoreCreateIfMissing, &pStore, NULL);
if (SUCCEEDED(hr)) {
    hr = m_pBasicDriver->Initialize(pStore);
}
```

В данном примере переменная `m_pWdfDevice` является указателем на интерфейс `IWDFDevice`, полученный драйвером при создании им инфраструктурного объекта устройства. Драйвер передает значение `NULL` в качестве имени хранилища свойств и значение `WdfPropertyStoreCreateIfMissing`, которое указывает инфраструктуре создать хранилище свойств с именем по умолчанию. Драйвер также передает значение `NULL` в параметре `pDisposition`, т. к. ему не требуется эта информация. Метод возвращает указатель на интерфейс `IWDFNamedPropertyStore` в переменной `pStore`.

В листинге 12.4 приводится пример получения драйвером значения именованного параметра из хранилища свойств. Данный код адаптирован из файла `Sideshow\BasicDDI.cpp`.

#### Листинг 12.4. Получение информации из хранилища свойств

```
PROPVARIANT pvBlob = {0};
PropVariantInit(&pvBlob);
hr = m_pPropertyStore->GetNamedValue(wszKey, &pvBlob);
if (SUCCEEDED(hr) && VT_BLOB == pvBlob.vt &&
    0 == (pvBlob.blob.cbSize % sizeof(APPLICATION_ID)))
{
    *pcAppIds = pvBlob.blob.cbSize/sizeof(APPLICATION_ID);
    *ppAppIds = (APPLICATION_ID*)pvBlob.blob.pBlobData;
}
```

В данном коде вызывается метод `IWDFNamedPropertyStore::GetNamedValue`, чтобы получить значение раздела, описываемого строкой `wszKey`. Это значение возвращается в виде двоичного значения вариантного типа `VT_BLOB`, над которым драйвер выполняет синтаксический разбор и оценку. Обратите внимание, что функция `PropVariantInit` является функцией COM, которая инициализирует структуру свойств вариантного типа.

Дополнительную информацию об этой функции см. в разделе **PropVariantInit** в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=79586>.

В листинге 12.5 приведен пример установки драйвером Sideshow значения именованного параметра в хранилище свойств. Этот код также адаптирован из файла `Sideshow\BasicDDI.cpp`.

#### Листинг 12.5. Внесение информации в хранилище свойств

```
PROPVARIANT pvBlob = {0};
PropVariantInit(&pvBlob);
pvBlob.vt = VT_BLOB;
pvBlob.blob.cbSize = cApps * sizeof(APPLICATION_ID);
pvBlob.blob.pBlobData = (BYTE*)pApps;
hr = m_pPropertyStore->SetNamedValue(wszKey, &pvBlob);
```

В листинге 12.5 образец драйвера записывает идентификатор приложения в хранилище свойств в виде двоичных данных. Переменной `PROPVARIANT` присваивается значение вариантного типа `VT_BLOB`, после чего вызывается метод `IWDFNamedPropertyStore::SetNamedValue`, чтобы сохранить это значение в разделе реестра, чье имя указано в строке `wszKey`.

## Объекты реестра KMDF и их методы

KMDF содержит многочисленные методы, посредством которых драйвер может выполнять операции чтения и записи реестра. С помощью этих методов драйвер может создавать, открывать и закрывать разделы реестра и опрашивать, модифицировать и удалять параметры разделов и отдельные элементы данных в них.

Драйвер может обращаться к элементу реестра, принадлежащему драйверу или устройству, как до, так и после создания объекта устройства. Если драйверу требуется информация из реестра до того, как он создаст объект устройства, он может получить значения отдельных свойств устройства или полностью извлечь раздел реестра для аппаратного обеспечения устройства (device hardware key) или раздел реестра для программного обеспечения драйвера (driver software key) с помощью методов семейства `WdfFdoInitXxx`. После создания объекта устройства драйвер применяет методы семейства `WdfDeviceXxx`. В табл. 12.4 приведена сводка методов для запроса отдельных свойств устройств и открытия разделов реестра.

**Таблица 12.4. Методы KMDF для запроса свойств и открытия разделов реестра**

Метод	Выполняемое действие
<code>WdfDeviceAllocAndQueryProperty</code>	По данному дескриптору объекта <code>WDFDEVICE</code> выделяет буфер и получает свойство устройства из реестра
<code>WdfDeviceQueryProperty</code>	По данному дескриптору объекта <code>WDFDEVICE</code> получает свойство устройства из реестра

Таблица 12.4 (окончание)

Метод	Выполняемое действие
WdfDeviceOpenRegistryKey	Открывает раздел реестра для аппаратного обеспечения устройства или раздел реестра для программного обеспечения драйвера и создает инфраструктурный объект раздела реестра, представляющий данный раздел реестра
WdfFdoInitAllocAndQueryProperty	По данному указателю на объект <code>WDFDEVICE_INIT</code> выделяет буфер и получает свойство устройства из реестра
WdfFdoInitOpenRegistryKey	По данному указателю на структуру <code>WDFDEVICE_INIT</code> открывает в реестре раздел аппаратного обеспечения устройства или раздел программного обеспечения драйвера и создает объект раздела реестра, представляющий данный раздел реестра
WdfFdoInitQueryProperty	По данному указателю на структуру <code>WDFDEVICE_INIT</code> получает свойство устройства из реестра

Чтобы прочитать значение раздела реестра, драйвер открывает раздел, после чего вызывает метод, запрашивающий данные из реестра. Для открытия раздела реестра применяются методы `WdfFdoInitOpenRegistryKey` и `WdfRegistryOpenKey`. Оба эти метода принимают следующие пять параметров:

- ◆ `DeviceInit` или `Device` — указатель на структуру `WDFDEVICE_INIT` для метода `WdfFdoInitOpenRegistryKey` или дескриптор объекта `WDFDEVICE` для метода `WdfRegistryOpenKey`;
- ◆ `DeviceInstanceKeyType` — значение типа `ULONG`, идентифицирующее раздел, который нужно открыть;
- ◆ `DesiredAccess` — битовая маска, указывающая тип требуемого доступа;
- ◆ `KeyAttributes` — необязательная структура атрибутов;
- ◆ `Key` — адрес, по которому получить дескриптор объекта `WDFKEY`.

Для получения или установки значения отдельного параметра в разделе применяются методы семейства `WdfRegistryXxx`, перечисленные в табл. 12.5. По окончании работы с реестром драйвер вызывает метод `WdfRegistryClose`, чтобы закрыть и удалить раздел.

Таблица 12.5. Методы KMDF для работы с разделами реестра

Метод	Описание
<code>WdfRegistryAssignMemory</code>	Назначает данные из буфера памяти именованному параметру в реестре
<code>WdfRegistryAssignMultiString</code>	Назначает набор строк из коллекции объектов строк именованному параметру в реестре
<code>WdfRegistryAssignString</code>	Назначает строку из объекта строки именованному параметру в реестре
<code>WdfRegistryAssignULong</code>	Назначает значение беззнакового длинного слова именованному параметру в реестре
<code>WdfRegistryAssignUnicodeString</code>	Назначает строку в кодировке Unicode именованному параметру в реестре

Таблица 12.5 (окончание)

Метод	Описание
WdfRegistryAssignValue	Назначает данные именованному параметру реестра
WdfRegistryClose	Закрывает раздел реестра, ассоциированный с объектом раздела реестра, после чего удаляет объект раздела реестра
WdfRegistryCreateKey	Создает и открывает раздел реестра, или просто открывает уже существующий раздел, и создает объект раздела реестра, представляющий данный раздел реестра
WdfRegistryOpenKey	Открывает раздел реестра и создает объект раздела реестра, представляющий данный раздел реестра
WdfRegistryQueryMemory	Получает данные из параметра реестра, сохраняет их в выделенный инфраструктурой буфер и создает объект памяти для представления буфера
WdfRegistryQueryMultiString	Получает строки, в настоящее время содержащиеся в многострочном параметре реестра, создает инфраструктурный объект строки для каждой строки и добавляет каждый объект строки в коллекцию
WdfRegistryQueryString	Получает строковые данные, в настоящее время содержащиеся в строковом параметре реестра, и присваивает данную строку объекту строки
WdfRegistryQueryULong	Получает данные беззнакового длинного слова (REG_DWORD), содержащиеся в настоящее время в параметре реестра, и копирует их в указанную драйвером область памяти
WdfRegistryQueryUnicodeString	Получает строковые данные, в настоящее время содержащиеся в строковом параметре реестра, и копирует данную строку в структуру UNICODE_STRING
WdfRegistryQueryValue	Получает данные, в настоящее время назначенные параметру реестра
WdfRegistryRemoveKey	Удаляет раздел реестра, ассоциированный с инфраструктурным объектом раздела реестра, после чего удаляет объект раздела реестра
WdfRegistryRemoveValue	Удаляет параметр и его данные из раздела реестра
WdfRegistryWdmGetHandle	Возвращает дескриптор WDM для раздела реестра, представляемого инфраструктурным объектом раздела реестра

В исходном файле Pcidrv.c образец драйвера PCIDRV предоставляет функции для выполнения следующих операций:

- ◆ чтения параметра реестра типа REG\_DWORD, записанного другим компонентом режима ядра или пользовательского режима;
- ◆ чтения параметра реестра типа REG\_DWORD, сохраненного в разделе устройства;
- ◆ записи параметра реестра типа REG\_DWORD, сохраненного в разделе устройства.

Функция `PciDrvReadFdoRegistryKeyValue` образца драйвера PCIDRV вызывается из функции обратного вызова `EvtDriverDeviceAdd` до создания драйвером объекта устройства. Она считывает раздел, записанный INF-файлом драйвера при установке, который указывает, был ли драйвер установлен как драйвер верхней кромки для мини-порта NDIS. Это важная инфор-

мация, т. к. она определяет, регистрирует ли драйвер определенные функции обратного вызова для событий политики энергопотребления и ввода/вывода. Если драйвер был установлен в качестве драйвера верхней кромки мини-порта, он не является менеджером политики энергопотребления для своего устройства и политика энергопотребления управляется согласно спецификации NDIS (Network Driver Interface Specification, спецификация интерфейса сетевых драйверов). Исходный код этой функции показан в листинге 12.6.

#### Листинг 12.6. Регистрация раздела реестра во время инициализации объекта устройства

```
BOOLEAN PciDrvReadFdoRegistryKeyValue(
    _in PWDFDEVICE_INIT           DeviceInit,
    _in PWCHAR                   Name,
    _out PULONG                  Value)
{
    WDFKEY      hKey = NULL;
    NTSTATUS    status;
    BOOLEAN     retVal = FALSE;
    UNICODE_STRING valueName;
    PAGED_CODE();
    *Value = 0;
    status = WdfFdoInitOpenRegistryKey(DeviceInit,
                                         PLUGPLAY_REGKEY_DEVICE,
                                         STANDARD_RIGHTS_ALL,
                                         WDF_NO_OBJECT_ATTRIBUTES,
                                         &hKey);
    if (NT_SUCCESS (status)) {
        RtInitUnicodeString (&valueName, Name);
        status = WdfRegistryQueryULong (hKey, &valueName, Value);
        if (NT_SUCCESS (status)) {
            retVal = TRUE;
        }
        WdfRegistryClose (hKey);
    }
    return retVal;
}
```

Первым делом драйвер инициализирует параметр `Value`, который получит значение запрошенного параметра раздела. Потом он открывает раздел реестра, вызывая метод `WdfFdoInitOpenRegistryKey`. Драйвер передает методу пять параметров, описанных ранее в этом разделе, и получает от него дескриптор возвращенного объекта `WDFKEY` в `hKey`. Константа `PLUGPLAY_REGKEY_DEVICE` указывает раздел аппаратного обеспечения устройства. Хотя образец драйвера запрашивает все права доступа к реестру, указывая `STANDARD_RIGHTS_ALL`, драйвер только читает раздел, но не записывает в него, поэтому можно было бы также указать `STANDARD_RIGHTS_READ`.

В случае успешного открытия драйвером раздела аппаратного обеспечения устройства, он запрашивает из раздела значение требуемого параметра. Имя параметра передается функции `PciDrvReadFdoRegistryKeyValue` в виде указателя на строку. Но для данного метода запроса KMDF требуется имя в виде счетной строки Unicode. Поэтому, прежде чем запросить значение, драйвер вызывает метод `RtInitUnicodeString`, чтобы скопировать входную строку в строку необходимого формата.

Потом драйвер вызывает метод `WdfRegistryQueryUlong`, который возвращает значение параметра раздела в параметре `Value`. Драйвер закрывает раздел, вызывая метод `WdfRegistryClose`, после чего функция возвращает управление.

## Общие объекты

*Общий объект* (general object), также называемый *базовым объектом* (base object), представляет собой определенный драйвером объект, поддерживающий подсчет ссылок, область контекста объекта, родительский объект, а также функцию обратного вызова удаления объекта. С помощью общего объекта драйвер может воспользоваться инфраструктурной моделью обратных вызовов и механизмом назначения родителей объектов для данных, не поддерживаемых никаким конкретным типом объекта WDF и не принадлежащим области контекста другого объекта.

Например, драйвер KMDF может использовать общий объект для управления временем жизни ресурса, используемого совместно несколькими рабочими элементами, процедурами DPC, или другими компонентами асинхронного исполнения. Драйвер создает общий объект, имеющий область контекста, а функция обратного вызова `EvtDestroyCallback` выделяет ресурс и сохраняет указатель на этот ресурс в области контекста. При каждом порождении драйвером асинхронного компонента исполнения драйвер получает ссылку на этот объект от имени модуля исполнения. Модуль исполнения освобождает ссылку по завершению работы с ресурсом. После порождения последнего модуля исполнения драйвер может удалить общий объект. Объект продолжает существовать до тех пор, пока последний модуль исполнения не освободит свою ссылку на него. Тогда инфраструктура вызывает функцию обратного вызова `EvtDestroyCallback`, которая освобождает ресурс.

Как драйверы UMDF, так и драйверы KMDF могут создавать общие объекты.

### Пример UMDF: создание общего объекта

Чтобы создать общий объект, драйвер UMDF создает объект обратного вызова со специфичной для драйвера функциональностью, после чего создает соответствующий инфраструктурный объект вызовом метода `IWDFDriver::CreateWdfObject`. Методу передаются следующие три параметра:

- ◆ `pCallbackInterface` — указатель на интерфейс `IUnknown` объекта обратного вызова;
- ◆ `pParentObject` — указатель на интерфейс `IWDFObject` родительского объекта, или `NULL`, чтобы принять объект "драйвер" родителем по умолчанию;
- ◆ `ppWdfObject` — указатель на буфер, принимающий указатель на интерфейс `IWDFObject` для созданного инфраструктурного объекта.

Инфраструктура применяет указатель на интерфейс `IUnknown` для запроса интерфейса `IObjectCleanup` на объекте обратного вызова. Если драйвер реализует интерфейс `IObjectCleanup`, то инфраструктура вызывает метод `OnCleanup`, чтобы известить драйвер непосредственно перед уничтожением объекта.

Драйвер может ассоциировать область контекста с инфраструктурным объектом или предоставить другой интерфейс `IObjectCleanup`, используя для этого метод `IWDFObject::AssignContext`.

В исходном коде в листинге 12.7 показывается пример создания драйвером UMDF общего объекта, для которого родителем является объект драйвера.

**Листинг 12.7. Создание общего объекта драйвером UMDF**

```

myObject = new CMyObject();
if (NULL == myObject){
    return E_OUTOFMEMORY;
}
hr = this->QueryInterface(_uuidof(IUnknown), (void **)unknown);
hr = pWdfDriver->CreateWdfObject(unknown, NULL, &fxWdfObject);
unknown->Release();
fxWdfObject->Release()

```

Сначала драйвер запрашивает указатель на интерфейс `IUnknown`. Потом он вызывает метод `CreateWdfObject`, передавая ему в параметрах указатель на интерфейс `IUnknown`, указатель `NULL`, чтобы принять объект драйвера в качестве родителя по умолчанию, и указатель на переменную, которая получает адрес интерфейса `IWdfObject` для созданного инфраструктурного объекта.

**Пример KMDF: создание общего объекта**

Драйвер KMDF создает общий объект, вызывая метод `WdfObjectCreate`, как показано в листинге 12.8.

**Листинг 12.8. Создание общего объекта драйвером KMDF**

```

WDF_OBJECT_ATTRIBUTES attributes;
WDFOBJECT myObject;
. . . // Код опущен.
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.EvtCleanupCallback = MyEvtCleanupCallback;
status = WdfObjectCreate(&attributes, &myObject);

```

Сначала драйвер объявляет переменные для структуры атрибутов и объекта. Потом он инициализирует структуру атрибутов, регистрирует функцию обратного вызова в этой структуре, после чего создает общий объект, вызывая метод `WdfObjectCreate`. Когда этот метод возвращает управление, переменная `myObject` содержит дескриптор созданного объекта. Перед тем, как удалить объект, инфраструктура вызывает функцию обратного вызова для очистки.

Для общего объекта драйвер KMDF может установить в структуре атрибутов объекта ограничение на исполнение только на уровне `passive`. Это ограничение заставляет инфраструктуру вызывать функции обратного вызова `EvtCleanupCallback` и `EvtDestroyCallback` драйвера на уровне `IRQL = PASSIVE_LEVEL`. Такая установка может быть полезной, если драйверу требуется ожидать завершения асинхронных операций перед очисткой объекта.

Уровни исполнения драйверов KMDF обсуждаются в главе 10.

**Объекты коллекции KMDF**

Объект коллекции — это связанный список объектов KMDF. Объекты коллекции могут быть все одного типа или разных типов. Объект может быть членом нескольких коллекций.

Коллекции полезны, когда драйверу нужно отслеживать несколько родственных объектов, особенно если объекты не требуют обслуживания в предсказуемом порядке.

Например:

- ◆ драйвер, который разбивает запрос на чтение или запись большого объема данных на несколько меньших запросов, создает коллекцию, содержащую все эти меньшие запросы ввода/вывода;
- ◆ драйвер, обрабатывающий запросы для нескольких устройств, создает коллекцию объектов устройств;
- ◆ драйвер устройства, поддерживающего прерывания MSI, создает коллекцию объектов прерывания, в которой каждый объект представляет одно сообщение.

Объекты коллекции имеют несколько преимуществ над простыми связанными списками, реализуемыми драйвером. А именно:

- ◆ при каждом добавлении драйвером элемента коллекции KMDF увеличивает счетчик ссылок этого элемента, поэтому дескриптор элемента остается действительным до тех пор, пока элемент находится в коллекции;
- ◆ KMDF реализует весь программный код, требуемый для отслеживания и управления списком, так что драйверам не требуется сохранять ссылку или индекс в области контекста каждого объекта.

Но по сравнению со связанными списками объекты коллекции имеют один существенный недостаток: если не имеется достаточно памяти, добавление объекта в коллекцию может завершиться неудачей. Поэтому драйверы всегда должны реализовывать проверку на такие ошибки. Обычно следует избегать применения объектов коллекции в критических ветвях, в которых нельзя допустить неудачного завершения исполнения.

## Методы коллекции

Объекты коллекции KMDF поддерживают методы для добавления и удаления элементов коллекции, возвращения элемента коллекции и возвращения числа элементов в коллекции. Методы объекта коллекции приводятся в табл. 12.6.

**Таблица 12.6. Методы объекта коллекции**

Метод	Действие
WdfCollectionAdd	Добавляет объект в коллекцию
WdfCollectionCreate	Создает объект коллекции
WdfCollectionGetCount	Возвращает число объектов в коллекции
WdfCollectionGetFirstItem	Возвращает дескриптор первого объекта в коллекции
WdfCollectionGetItem	Возвращает дескриптор объекта в коллекции по его индексу
WdfCollectionGetLastItem	Возвращает дескриптор последнего объекта в коллекции
WdfCollection Remove	Удаляет объект из коллекции по его дескриптору
WdfCollectionRemoveItem	Удаляет объект из коллекции по его индексу в коллекции

KMDF не предоставляет никаких возможностей синхронизации для объекта коллекции, поэтому, если к коллекции могут одновременно обращаться несколько функций драйвера, драйвер должен создавать и захватывать собственную блокировку для объекта коллекции.

Объекты коллекции проиндексированы, начиная с нуля, и драйвер может извлечь и удалить объект, указывая его индекс. Драйвер может также удалить объект по его дескриптору. Когда инфраструктура удаляет объект из коллекции, она корректирует индексы должным образом. Например, если драйвер удаляет *n*-й объект из коллекции, объект *n+1* становится объектом номером *n* и т. д.

Инфраструктура увеличивает значение счетчика ссылок объекта, когда драйвер добавляет объект в коллекцию и уменьшает его, когда драйвер удаляет объект из коллекции.

## Пример: создание и использование коллекции

Образец драйвера Toastmon использует коллекцию для хранения информации об устройствах Toaster подключенных к системе. При добавлении каждого интерфейса устройства Toaster драйвер создает соответствующий объект получателя ввода/вывода и добавляет его в коллекцию получателей ввода/вывода Toaster. При удалении каждого интерфейса устройства Toaster драйвер удаляет его из коллекции. Так как к коллекции обращается несколько функций драйвера, драйвер создает wait-блокировку WDF для сериализации доступа к коллекции. Драйвер никогда не обращается к коллекции на уровне IRQL  $\geq$  DISPATCH\_LEVEL, поэтому в spin-блокировке нет надобности.

В листинге 12.9 приведен пример создания драйвером Toastmon коллекции и блокировки. Код взят из файла Toaster\Toastmon\Toastmon.c образца драйвера Toastmon.

### Листинг 12.9. Создание объекта коллекции драйвером KMDF

```
WDF_OBJECT_ATTRIBUTES attributes;
NTSTATUS status = STATUS_SUCCESS;
WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
attributes.ParentObject = device;
status = WdfCollectionCreate(&attributes,
                            &deviceExtension->TargetDeviceCollection);
if (!NT_SUCCESS(status)) {
    . . . // Код для обработки ошибки опущен.
}
WF_OBJECTJUTRIBUTES_INIT(&attributes);
attributes.ParentObject = device;
status = WdfWaitLockCreate(&attributes,
                           &deviceExtension->TargetDeviceCollectionLock);
if (!NT_SUCCESS(status)) {
    . . . // Код для обработки ошибки опущен.
}
```

В листинге 12.9 переменная `device` содержит дескриптор объекта `WDFDEVICE`, а переменная `deviceExtension` — указатель на область контекста для объекта устройства. Область контекста определяется следующим образом:

```
typedef Struct _DEVICE_EXTENSION {
    WDFDEVICE WdfDevice;
    WDFIOTARCKET ToasterTarget;
    PVOID NotificationHandle; // Дескриптор извещения интерфейса
    WDFCOLLECTION TargetDeviceCollection;
```

```

WDFWAITLOCK      TargetDeviceCollectionLock;
 PVOID            WMIDeviceArrivalNotificationObject;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

```

Прежде чем создать объект коллекции, драйвер инициализирует структуру атрибутов объекта и устанавливает объект устройства в качестве родителя коллекции. Объект коллекции создается вызовом метода `WdfCollectionCreate`, который возвращает дескриптор объекта в поле `TargetDeviceCollection` области контекста объекта устройства.

Подобным образом создается и блокировка. Драйвер инициализирует структуру атрибутов объекта, присваивает полю `Parent` объект устройства и вызывает метод `WdfWaitLockCreate`, чтобы создать объект wait-блокировки. Драйвер сохраняет дескриптор объекта блокировки в поле `TargetDeviceCollectionLock` области контекста объекта устройства.

Дополнительную информацию по wait-блокировкам см. в главе 10.

Когда подключается новый интерфейс устройства, драйвер `Toastmon` создает объект получателя ввода/вывода, представляющий этот интерфейс. В листинге 12.10 показано, каким образом драйвер добавляет объект получателя ввода/вывода в коллекцию.

#### Листинг 12.10. Добавление объекта в коллекцию

```

WdfWaitLockAcquire(deviceExtension->TargetDeviceCollectionLock, NULL);
status = WdfCollectionAdd(deviceExtension-> TargetDeviceConection,
                           ioTarget);
if (!NT_SUCCESS(status)) {
    . . . // Код для обработки ошибки опущен.
}
WdfWaitLockRelease(deviceExtension->TargetDeviceCollectionLock);

```

Драйвер захватывает блокировку для защиты коллекции, вызывая метод `WdfWaitLockAcquire`, передавая `NULL` для значения тайм-аута, таким образом указывая, что он будет дожидаться блокировку неограниченное время. Потом драйвер добавляет объект в коллекцию, вызывая для этого метод `WdfCollectionAdd` и передавая ему дескриптор коллекции и дескриптор объекта. По окончании обращения к коллекции драйвер освобождает блокировку, вызывая метод `WdfWaitLockRelease`.

В листинге 12.11 приведен пример удаления драйвером объекта из коллекции.

#### Листинг 12.11. Удаление объекта из коллекции

```

WdfWaitLock<Acquire(deviceExtension->TargetDeviceCollectionLock, NULL);
WdfCollectionRemove(deviceExtension-> TargetDeviceCollection, IoTarget);
WdfWaitLockRelease(deviceExtension-> TargetDeviceCollectionLock);

```

Как можно видеть, удаление объекта не представляет ничего сложного: захватывается блокировка, вызывается метод `WdfCollectionRemove` и освобождается блокировка. Драйвер может удалить объект из коллекции, указывая дескриптор объекта, как это делается в примере, или указывая индекс объекта в коллекции.

Драйвер не удаляет коллекцию, т. к. она автоматически удаляется инфраструктурой при удалении ею объекта устройства, который является родителем объекта коллекции.

## Объекты таймера KMDF

Драйвер KMDF может использовать объект таймера для периодической активации функции обратного вызова или для ее одноразовой активации по истечении указанного периода времени. Драйверы применяют таймеры для разнообразных целей. Например:

- ◆ образец драйвера Pcidrv создает таймер WDT для обнаружения соединения в своем устройстве и для проверки на наличие зависшего оборудования;
- ◆ образец драйвера Serial создает несколько таймеров для ограничения времени выполнения своих операций чтения и записи;
- ◆ образец драйвера Toastmon создает периодически срабатывающий таймер для отправления запросов своему получателю ввода/вывода через равные промежутки времени.

Каждый объект таймера ассоциирован с функцией обратного вызова `EvtTimerFunc`. По истечении таймера инфраструктура добавляет эту функцию в системную очередь процедур DPC.

Если устройство переводится в пониженное состояние энергопотребления или останавливается для перераспределения ресурсов, инфраструктура не останавливает таймер. Драйвер может остановить таймер в функции обратного вызова самоуправляемого ввода/вывода для соответствующего события. Если устройство удаляется, то перед удалением объекта таймера инфраструктура останавливает таймер.

Пример использования функций обратного вызова самоуправляемого ввода/вывода для управления таймером приводится в главе 8.

### Методы объекта таймера

В табл. 12.7 приведен список методов, поддерживаемых объектом таймера.

Для создания объекта таймера драйвер вызывает метод `WdfTimerCreate`, передавая ему указатель на структуру `WDF_TIMER_CONFIG` и указатель на WDF-структуре `WDF_OBJECT_ATTRIBUTES`.

**Таблица 12.7. Методы объекта таймера**

Метод	Действие
<code>WdfTimerCreate</code>	Создает объект таймера
<code>WdfTimerGetParentObject</code>	Возвращает родителя объекта таймера
<code>WdfTimerStart</code>	Запускает таймер
<code>WdfTimerStop</code>	Останавливает таймер

Информация о таймере указывается в следующих полях структуры `WDF_TIMER_CONFIG`:

- ◆ `EvtTimerFunc` — указатель на поставляемую драйвером функцию обратного вызова `EvtTimerFunc`;
- ◆ `Period` — период времени в миллисекундах (мс). Инфраструктура вызывает функцию `EvtTimerFunc` периодически по истечении указанного периода времени. Если это значение равно нулю, то инфраструктура вызывает функцию обратного вызова только один раз — по истечении периода времени, указанного в параметре `DueTime` метода `WdfTimerStart`. Значение периода времени не может быть отрицательным;

- ◆ `AutomaticSerialization` — булево значение. Если равно `TRUE`, то инфраструктура будет синхронизировать исполнение функции обратного вызова `EvtTimerFunc` объекта таймера с функциями обратного вызова других объектов, являющимися потомками родительского объекта устройства таймера. Область синхронизации для родительского объекта устройства должна быть `WdfSynchronizationScopeDevice` или `WdfSynchronizationScopeQueue`.

Для инициализации структуры конфигурации таймера инфраструктура предоставляет следующие две функции.

- ◆ Функция `WDF_TIMER_CONFIG_INIT` организует структуру конфигурации для таймера с одноразовым срабатыванием.

В качестве параметров функции передаются указатель на структуру `WDF_TIMER_CONFIG` и указатель на функцию обратного вызова `EvtTimerFunc`, которую инфраструктура должна поставить в очередь по истечении таймера.

- ◆ Функция `WDF_TIMER_CONFIG_INIT_PERIODIC` организовывает структуру конфигурации для таймера с периодическим срабатыванием.

В качестве параметров функция принимает значение, указывающее интервал срабатывания таймера и те же самые параметры, что и функция `WDF_TIME_CONFIG_INIT`.

Обе функции устанавливают значение поля `AutomaticSerialization` структуры в `TRUE`.

Дополнительную информацию об автоматической сериализации см. в главе 10.

## Временные интервалы

Время для поля `Period` структуры конфигурации таймера и для параметра `DueTime` функции `WdfTimerStart` указывается в разных единицах.

- ◆ Для поля `Period` время указывается в миллисекундах.
- ◆ Время параметра `DueTime` метода `WdfTimerStart` указывается в системных единицах времени, представляющих собой интервалы длиной в 100 наносекунд.

Значение параметра `DueTime` может быть либо абсолютным временем, либо относительным к текущему системному времени. Для периодического таймера параметр `DueTime` указывает первое срабатывание таймера. Интервал последующих срабатываний определяется значением поля `Period`.

- Абсолютное значение времени представляет собой положительное значение, указывающее количество интервалов длиной в 100 наносекунд, прошедших с 00:00 часов 1 января 1601 г.
- Значение относительного времени представляет разницу с текущим системным временем. Значения относительного времени указываются отрицательными числами.

В KMDF имеется несколько функций для преобразования из одного вида значений времени в другой. Эти функции для преобразования типов периодов времени можно также использовать для указания значений тайм-аута для запросов ввода/вывода. В табл. 12.8 приведен список функций для преобразований между типами периодов времени.

**Таблица 12.8. Функции для преобразования между типами периодов времени**

Функция	Описание
<code>WDF_ABS_TIMEOUT_IN_MS</code>	Преобразовывает значение в миллисекундах в абсолютное время в системных единицах времени

Таблица 12.8 (окончание)

Функция	Описание
WDF_ABS_TIMEOUT_IN_SEC	Преобразовывает значение в секундах в абсолютное время в системных единицах времени
WDF_ABS_TIMEOUT_IN_US	Преобразовывает значение в микросекундах в абсолютное время в системных единицах времени
WDF_REL_TIMEOUT_IN_MS	Преобразовывает значение в миллисекундах в относительное время в системных единицах времени
WDF_REL_TIMEOUT_IN_SEC	Преобразовывает значение в секундах в относительное время в системных единицах времени
WDF_REL_TIMEOUT_IN_US	Преобразовывает значение в микросекундах в относительное время в системных единицах времени

Например, чтобы установить время срабатывания таймера в 5 секунд, драйвер передает методу `WdfTimerStart` функцию `WDF_REL_TIMEOUT_IN_SEC(5)`.

## Функция обратного вызова *EvtTimerFunc*

Функция обратного вызова является процедурой DPC, исполняющейся на уровне `IRQL = DISPATCH_LEVEL`. По истечении таймера инфраструктура добавляет эту функцию в конец системной очереди процедур DPC; функция исполняется, когда она достигает начала очереди.

Прототип функции *EvtTimerFunc* следующий:

```
typedef VOID
    (*PVOID_WDF_TIMER) (
        IN WDFTIMER Timer
    );
```

Здесь `Timer` является дескриптором объекта таймера, активировавшего обратный вызов.

Так как функция исполняется на уровне `DISPATCH_LEVEL`, она не должна выполнять никаких действий, которые могут вызвать страничную ошибку. Если функции таймера нужно выполнить действия, требующие исполнения на уровне `PASSIVE_LEVEL`, то для их исполнения функция должна создать объект рабочего элемента и поставить в очередь ассоциированную с ним функцию обратного вызова.

Дополнительную информацию о рабочих элементах и правилах исполнения на уровне `DISPATCH_LEVEL` см. в главе 15.

## Пример: использование объекта таймера

Образец драйвера `Toastmon` использует таймер для отправления периодических запросов каждому получателю ввода/вывода драйвера. При создании каждого объекта получателя ввода/вывода драйвер также создает таймер.

В листинге 12.12 показано, каким образом драйвер создает и запускает таймер. Код примера, приведенного в листинге, взят из файла `Toastmon.c`.

**Листинг 12.12. Создание и запуск объекта таймера**

```

NTSTATUS status = STATUS_SUCCESS;
PTARGET_DEVICE_INFO targetDeviceInfo = NULL;
WDFIOTARGET ioTarget;
WDF_OBJECT_ATTRIBUTES attributes;
WDF_TIMER_CONFIG wdfTimerConfig;
// Создаем периодически срабатывающий таймер для
// отправления запросов получателю ввода/вывода.
WDF_TIMER_CONFIG_INIT_PERIODIC(&wdfTimerConfig,
                               Toastmon_EvtTimerPostRequests,
                               PERIODIC_TIMER_INTERVAL); //ms
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, TIMER_CONTEXT);
// Устанавливаем IoTarget в качестве родителя таймера.
attributes.ParentObject = ioTarget;
targetDeviceInfo = GetTargetDeviceInfo(ioTarget);
status = WdfTimerCreate(&wdfTimerConfig,
                       &attributes,
                       &targetDeviceInfo->TimerForPostingRequests);
if (!NT_SUCCESS(status)) {
    WdfObjectDelete(ioTarget);
    return status;
}
GetTimerContext(targetDeviceInfo->
                TimerForPostingRequests)->IoTarget = ioTarget;
// Запускаем таймер.
WdfTimerStart(targetDeviceInfo->TimerForPostingRequests,
              WDF_REL_TIMEOUT_IN_MS(1));

```

Как можно видеть в листинге 12.12, для конфигурации периодического таймера драйвер вызывает метод `WDF_TIMER_CONFIG_INIT_PERIODIC`. В качестве параметров методу передается указатель на структуру `WDF_TIMER_CONFIG`, указатель на функцию обратного вызова `EvtTimerFunc` драйвера и константа `PERIODIC_TIMER_INTERVAL`, которую драйвер определил в заголовочном файле `Toastmon.h` как 1000.

После этого драйвер инициализирует структуру атрибутов для объекта таймера, указывая тип области контекста для таймера. Драйвер использует область контекста объекта таймера для хранения дескриптора объекта получателя ввода/вывода, с которым ассоциирован таймер. Драйвер присваивает полю `ParentObject` структуры атрибутов значение получателя ввода/вывода, поэтому объект получателя ввода/вывода не может быть удален, пока используется функция обратного вызова `EvtTimerFunc`.

Дальше указатель получает указатель на область контекста для объекта получателя ввода/вывода, вызывая функцию доступа `GetTargetDeviceInfo` драйвера. Потом драйвер вызывает метод `WdfTimerCreate`, передавая ему в качестве параметров инициализированную структуру конфигурации, инициализированную структуру атрибутов и указатель на адрес в области контекста объекта получателя ввода/вывода, в котором драйвер хранит дескриптор объекта таймера. Если метод `WdfTimerCreate` завершается неудачей, драйвер удаляет объект получателя ввода/вывода.

Драйвер сохраняет дескриптор объекта получателя ввода/вывода в области контекста объекта таймера, после чего заполняет таймер. В качестве параметров метод `WdfTimerStart` принимает дескриптор объекта таймера и время исполнения в системных единицах времени.

Драйвер указывает время исполнения в 1 мс, в результате чего инфраструктура первый раз ставит функцию *EvtTimerFunc* в очередь после истечения 1 мс. Впоследствии функция ставится в очередь каждую секунду (1000 мс), как было указано драйвером в структуре конфигурации. Драйвер применяет для первого запуска таймера интервал в 1 мс, потому что это самый короткий интервал, с которым может работать Windows.

Драйвер останавливает таймер, когда инфраструктура извещает его о запросе на удаление по результатам опроса или о запросе на удаление для получателя ввода/вывода. В листинге 12.13 показано, каким образом драйвер останавливает и удаляет таймер.

#### Листинг 12.13. Остановка и удаление объекта таймера

```
targetDeviceInfo = GetTargetDeviceInfo(IoTarget);
WdfTimerStop(targetDeviceInfo->TimerForPostingRequests, TRUE);
WdfWorkItemFlush(targetDeviceInfo->WorkItemForPostingRequests);
WdfObjectDelete(IoTarget);
```

Драйвер вызывает метод *WdfTimerStop*, передавая ему в параметрах дескриптор объекта таймера и булево значение. Если булево значение равно TRUE, то метод *WdfTimerStop* не должен возвращать управление до тех пор, пока не завершится исполнение всех поставленных в очередь функций *EvtTimerFunc* и любых других процедур DPC драйвера. Функция обратного вызова создает рабочий элемент для выполнения обработки на уровне PASSIVE\_LEVEL. Поэтому после возвращения управления методом *WdfTimerStop* рабочие элементы больше нельзя ставить в очередь, так что драйвер может сбросить очередь рабочих элементов. Потом образец драйвера удаляет объект получателя ввода/вывода, который является родителем как таймера, так и рабочего элемента.

Когда драйвер удаляет получатель ввода/вывода, инфраструктура ожидает завершения таймера и рабочих элементов, поэтому остановка драйвером таймера и сброс рабочих элементов может казаться излишним. Но при удалении родителя инфраструктура не гарантирует порядок удаления его множественных потомков. Поэтому драйвер явно останавливает таймер и сбрасывает очередь, чтобы инфраструктура могла безопасно удалить эти два объекта в любом порядке.

## Поддержка интерфейса WMI в драйвере KMDF

Интерфейс WMI предоставляет драйверам способ для экспортирования информации другим компонентам. Обычно интерфейс WMI применяется в драйверах для следующих целей:

- ◆ разрешить приложениям пользовательского режима запрашивать связанную с устройством информацию, например, данные о производительности;
- ◆ разрешить администратору, имеющему соответствующие привилегии, управлять устройством, исполняя приложение на удаленной системе.

KMDF поддерживает интерфейс WMI для объектов устройства Plug and Play через функции обратного вызова и типы объектов, специфичных для WMI. В оставшемся материале этой главы рассматривается встроенная в KMDF поддержка для интерфейса WMI.

## Работа с WMI

Драйвер KMDF, поддерживающий WMI, регистрируется как поставщик информации WMI и регистрирует один или несколько экземпляров этой информации. Каждый поставщик

WMI ассоциирован с определенным GUID. Для потребления данных из экземпляра данных поставщика WMI другие компоненты регистрируются с таким же само GUID. Компоненты пользовательского режима запрашивают данные экземпляра WMI, вызывая функции COM, которые система преобразовывает в запросы `IRP_MJ_SYSTEM_CONTROL` и отправляет целевым поставщикам.

KMDF поддерживает запросы WMI посредством своего обработчика запросов WMI, который предоставляет драйверам следующие возможности.

- ◆ Реализацию WMI по умолчанию.

Драйверы, не предоставляющие данных WMI, не должны регистрироваться в качестве поставщиков WMI; все запросы `IRP_MJ_SYSTEM_CONTROL` обрабатываются инфраструктурой KMDF.

- ◆ Функции обратного вызова на отдельных экземплярах, а не только на уровне объекта устройства, так что разные экземпляры могут работать по-разному.
- ◆ Проверку достоверности размеров буферов, чтобы обеспечить, что размеры буферов, используемых в запросах WMI, отвечают требованиям ассоциированного поставщика и экземпляра.

Стандартная реализация WMI содержит поддержку флагжков (check box) на вкладке **Power Management** (Управление энергопотреблением) Диспетчера устройств. С помощью флагжков пользователь может управлять возможностью устройства пробуждать систему, а также возможностью перевода устройства системой в режим пониженного энергопотребления при его простое. Если драйвер разрешает эту возможность в своих настройках политики энергопотребления, то KMDF обрабатывает такие запросы автоматически.

Когда KMDF получает запрос `IRP_MJ_SYSTEM_CONTROL`, предназначенный для драйвера KMDF, она предпринимает следующие действия:

- ◆ если драйвер зарегистрировался как поставщик WMI и зарегистрировал один или несколько экземпляров WMI, обработчик WMI активирует функции обратного вызова для этих экземпляров соответствующим образом;
- ◆ если драйвер не зарегистрировал никаких экземпляров WMI, обработчик WMI отвечает на запрос, предоставив запрошенные данные, если он может, передавая запрос следующему нижнему драйверу или завершая запрос неудачей.

Точно так же, как и объекты WDF, объекты экземпляров WMI имеют область контекста. Драйвер может использовать область контекста объекта экземпляра WMI в качестве источника данных, доступного только для чтения, таким образом, позволяя сбор данных при минимальных усилиях. Драйвер может удалить объекты экземпляров WMI в любое время после их создания.

Функции обратного вызова WMI не синхронизируются с состояниями Plug and Play и энергопотребления устройства. Поэтому, когда происходит событие WMI, KMDF вызывает функции обратного вызова драйвера для WMI, даже если устройство и не находится в рабочем состоянии.

## Требования для поддержки WMI

Драйвер KMDF поддерживает WMI, выполняя одно из следующих требований:

- ◆ инициализирует свою поддержку WMI, регистрируя поставщика данных WMI и создавая один или больше объектов экземпляра WMI для представления блоков данных, которые он может считывать или записывать;

- ◆ дополнительно реализует одну или несколько функций обратного вызова для предоставления данных WMI, поставляемых драйвером;
- ◆ дополнительно активирует события WMI.

В своих функциях обратного вызова для WMI драйвер может вызывать методы WMI на объекте устройства для создания и манипулирования экземплярами WMI или для изменения своего статуса поставщика WMI. После возвращения управления функциями обратного вызова WMI, инфраструктура завершает или пересыпает запрос от имени драйвера, в зависимости от того, которое из этих действий является уместным.

Информацию о возможностях WMI доступных в режиме ядра см. в разделе **Windows Management Instrumentation** (Инструментарий управления Windows) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81581>. Также дополнительную информацию о поддержке WMI в KMDF см. в разделе **Supporting WMI in Framework-based Drivers** (Поддержка WMI в драйверах, основанных на инфраструктуре WDF) по адресу <http://go.microsoft.com/fwlink/?LinkId=82325>.

## Инициализация поддержки WMI

Чтобы инициализировать свою поддержку для WMI, драйвер KMDF выполняет следующие три шага, обычно в своей функции обратного вызова `EvtDriverDeviceAdd` или `EvtDeviceSelfManagedToInit`:

1. Регистрирует имя ресурса MOF драйвера (Managed object format, формат управляемых объектов).
2. Инициализирует структуру конфигурации поставщика WMI и создает объект поставщика WMI.
3. Инициализирует структуру конфигурации экземпляра WMI и создает экземпляр WMI.

### Ресурс MOF

Ресурс MOF находится в файле сценария ресурсов (с фрасширением RE) драйвера и определяет данные WMI и блоки событий. Драйвер KMDF регистрирует имя ресурса MOF, вызывая метод `WdfDeviceAssignMofResourceName`.

### Объект поставщика WMI

После регистрации драйвером своего ресурса MOF, он может инициализировать структуру конфигурации поставщика WMI (`WDF_WMI_PROVIDER_CONFIG`) следующей информацией:

- ◆ GUID поставщика;
- ◆ один или несколько флагов, указывающих, реализует ли поставщик функции обратного вызова, специфичные для данного экземпляра, требуют ли процедуры сбора данных повышенной производительности, а также поддерживает ли поставщик WMI трассировку событий;
- ◆ размер буфера для экземпляров поставщика;
- ◆ указатель на функцию обратного вызова `EvtWmiProviderFunctionControl` драйвера, если драйвер реализует эту функцию.

В структуре `WDF_WMI_PROVIDER_CONFIG` драйвер указывает размер буфера, требуемого для функций обратного вызова `EvtWmiInstanceQueryInstance` и `EvtWmiInstanceSetInstance` по-

ставщика. Если драйвер задает такое значение, по прибытию запроса WMI KMDF проверяет размер буфера на достоверность и активизирует функции обратного вызова, только если предоставлен буфер достаточного размера. Если размер экземпляра динамический или отсутствует при создании поставщика, драйвер не может сконфигурировать размер буфера. Вместо этого драйвер должен указать ноль для значения этого поля и размер буферов должен проверяться самими функциями обратного вызова.

Функция обратного вызова *EvtWmiProviderFunctionControl* включает и выключает сбор данных WMI для драйвера и является необязательной. Драйверы, собирающие большие объемы данных для конкретного блока, должны реализовывать эту функцию и устанавливать *WdfWmiProviderExpensive*-флаг в поле *Flags* структуры *WDF\_WMI\_PROVIDER\_CONFIG*.

После заполнения драйвером структуры конфигурации он может создать объект поставщика WMI. Если драйверу требуется только один поставщик WMI, в вызове метода для создания объекта нет надобности. Чтобы упростить реализацию таких драйверов, KMDF создает поставщика WMI по умолчанию, когда драйвер создает свой первый экземпляр WMI. Поэтому драйвер должен вызывать метод *WdfWmiProviderCreate* только в тех случаях, когда ему требуется больше чем один поставщик WMI.

## Объекты экземпляра WMI

Дальше драйвер создает объекты экземпляра WMI для поставщика. Для каждого экземпляра WMI драйвер заполняет структуру конфигурации типа *WDF\_WMI\_INSTANCE\_CONFIG*. Эта структура содержит следующую информацию.

- ◆ Дескриптор поставщика WMI и указатель на структуру конфигурации поставщика.
- ◆ Булево значение в поле *UseContextForQuery*. При значении этого поля **TRUE** драйвер просто сохраняет запрошенные данные, предназначенные только для чтения, в области контекста объекта экземпляра WMI.

По умолчанию значение этого поля равно **FALSE**, указывая, что драйвер предоставляет функцию обратного вызова по событию для запросов за данными к экземплярам.

- ◆ Булево значение в поле *Register*. При значении этого поля **TRUE** KMDF должна регистрировать экземпляр поставщика драйвера в WMI.

По умолчанию значение этого поля равно **FALSE**, указывая, что драйвер вызовет метод *WdfWmiInstanceRegister* самостоятельно.

- ◆ Указатели на функции обратного вызова для событий *EvtWmiInstanceQueryInstance*, *EvtWmiInstanceSetInstance*, *EvtWmiInstanceSetItem* и *EvtWmiInstanceExecuteMethod*, если драйвер реализует эти функции.

Если драйвер имеет только один экземпляр WMI, поставляющий данные фиксированного размера, предназначенные только для чтения, со своей области контекста объекта, то в функциях обратного вызова нет надобности.

Если драйвер уже получил дескриптор объекта поставщика, вызвав метод *WdfWmiProviderCreate*, то он может использовать функцию *WDF\_WMI\_INSTANCE\_CONFIG\_INIT\_PROVIDER*, чтобы сохранить дескриптор поставщика и конфигурационную информацию в структуре конфигурации экземпляра. Если драйвер не имеет этого дескриптора, вместо функции *WDF\_WMI\_INSTANCE\_CONFIG\_INIT\_PROVIDER* он должен использовать функцию *WDF\_WMI\_INSTANCE\_CONFIG\_INIT\_PROVIDER\_CONFIG*, которая записывает информацию о поставщике, которого KMDF создала для драйвера.

Когда структура конфигурации для экземпляра заполнена должным образом, драйвер вызывает метод `WdfWmiInstanceCreate`, чтобы создать экземпляр.

### Пример: пример инициализации поддержки WMI

Модуль `Wmi.c` образца драйвера `Featured Toaster` содержит код для реализации сбора данных WMI. Этот код помещен в драйвер `Featured Toaster` единственно для демонстрационных целей.

Код для регистрации WMI содержится во вспомогательной функции `ToasterWmiRegistration`, которая вызывается функцией обратного вызова `ToasterEvtDeviceAdd`. Исходный код этой функции показан в листинге 12.14.

#### Листинг 12.14. Инициализация поддержки WMI

```
NTSTATUS ToasterWmiRegistration(WDFDEVICE Device)
{
    NTSTATUS status;
    PFDO_DATA fdoData;
    PToasterDeviceInformation pData;
    PToasterControl controlData;
    WDFWMIINSTANCE instance;
    WDF_OBJECT_ATTRIBUTES woa;
    WDF_WMI_PROVIDER_CONFIG providerConfig;
    WDF_WMI_INSTANCE_CONFIG instanceConfig;
    DECLARE_CONST_UNICODE_STRING(mofRsrcName, MOFRESOURCENAME);
    PAGED_CODE();
    fdoData = ToasterFdoGetData(Device);
    status = WdfDeviceAssignMofResourceName(Device, &mofRsrcName);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    WDF_WMI_PROVIDER_CONFIG_INIT(&providerConfig,
        &ToasterDeviceInformation_CUID);
    providerConfig.MinInstanceIdBufferSize = 0;
    WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG(&instanceConfig,
        &providerConfig);
    instanceConfig.Register = TRUE;
    instanceConfig.EvtWmiInstanceQueryInstance =
        EvtWmiInstanceStdDeviceDataQueryInstance;
    instanceConfig.EvtWmiInstanceSetInstance =
        EvtWmiInstanceStdDeviceDataSetInstance;
    instanceConfig.EvtWmiInstanceSetItem =
        EvtWmiInstanceStdDeviceDataSetItem;
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&woa,
        ToasterDeviceInformation);
    status = WdfWmiInstanceCreate(Device, &instanceConfig, &woa,
        &instance);
    if (!NT_SUCCESS(status)) {
        return status;
    }
    pData = ToasterWmiGetData(instance);
    pData->ConnectorType = TOASTER_WMI_STD_USB;
    pData->Capacity = 2000;
```

```
pData->ErrorCount = 0;
pData->Controls = 5;
pData->DebugPrintLevel = DebugLevel;
WDF_WMI_PROVIDER_CONFIG_INIT(&providerConfig,
    &TOASTER_NOTIFY_DEVICE_ARRIVAL_EVENT);
providerConfig.Flags = WdfWmiProviderEventOnly;
providerConfig.MinInstanceBufferSize = 0;
WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG(&instanceConfig,
    &providerConfig);
instanceConfig.Register = TRUE;
status = WdfWmiInstanceCreate (Device, &instanceConfig,
    WDF_NO_OBJECT_ATTRIBUTES, &fdoData->WmiDeviceArrivalEvent
);
if (!NT_SUCCESS(status)) {
    return status;
}
. . . // Дополнительный код опущен.
return status;
}
```

Образец драйвера присваивает имя своему ресурсу MOF, вызывая метод `WdfDeviceAssignMofResourceName`, передавая ему в параметрах дескриптор объекта устройства и указатель на имя. Имя определено как "ToasterWMI" в заголовочном файле `Toaster.h`, который находится в папке `Toaster\Func\Shared`.

Дальше драйвер инициализирует структуру `WDF_WMI_PROVIDER_CONFIG`, чтобы он мог зарегистрироваться в качестве поставщика данных WMI. Для этого он вызывает функцию `WDF_WMI_PROVIDER_CONFIG_INIT`, передавая ей указатель на структуру конфигурации и указатель на GUID, который нужно ассоциировать с его блоками данных WMI. После этого он устанавливает минимальный размер буфера в структуре конфигурации поставщика. Установка нулевого размера буфера означает, что проверка достоверности размера буфера выполняется функциями обратного вызова драйвера, а не инфраструктурой. Драйвер принимает установки флагов по умолчанию и не указывает функцию обратного вызова `EvtWmiProviderFunctionControl`.

Дальше драйвер выполняет конфигурацию первых двух экземпляров WMI. Он вызывает функцию `WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG`, чтобы создать структуру конфигурации экземпляра WMI, которая ассоциируется со структурой конфигурации поставщика. Чтобы указать, что, кроме создания экземпляра, KMDF также должна зарегистрировать драйвер в качестве поставщика, он устанавливает значение поля `Register` структуры в `TRUE`. В структуре конфигурации экземпляра драйвер регистрирует функции обратного вызова для трех событий WMI: `EvtWmiInstanceQueryInstance`, `EvtWmiInstanceSetInstance` и `EvtWmiInstanceSetItem`.

Драйвер использует область контекста объекта экземпляра для хранения некоторых данных WMI. Блок данных содержит строку переменной длины, поэтому для возврата данных драйвер должен реализовать функцию обратного вызова `EvtWmiInstanceQueryInstance`. Чтобы установить тип области контекста как `ToasterDeviceInformation`, драйвер вызывает макрос `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE`.

После инициализации драйвером структуры конфигурации и установки атрибутов для объекта экземпляра WMI, он вызывает метод `WdfWmiInstanceCreate`, передавая ему в качестве параметров дескриптор объекта устройства и указатели на структуры конфигурации и атри-

бутов. Этот метод создает экземпляр WMI и возвращает дескриптор этого экземпляра по адресу `&instance`.

Дальше драйвер вызывает функцию доступа `ToasterWmiGetData`, чтобы получить указатель на область контекста объекта, после чего сохраняет специфичные для драйвера данные в этой области контекста.

Этим завершается процесс создания и инициализации первого экземпляра WMI. Дальше драйвер создает второй объект экземпляра WMI и поставщика для активации событий. Он повторно инициализирует структуру конфигурации поставщика, на этот раз передавая указатель на GUID для события подключения устройства и устанавливая флаг `WdfWmiProviderEventOnly`. Этот флаг означает, что клиенты этого экземпляра могут регистрироваться для получения событий, но они не могут запрашивать или устанавливать данные экземпляра. Драйвер снова задает размер буфера, после чего конфигурирует и создает экземпляр, выполняя ту же последовательность вызовов функций и методов, что и при создании первого экземпляра.

## Функции обратного вызова для событий экземпляра WMI

Драйверы KMDF могут реализовывать функции обратного вызова для следующих событий экземпляра WMI:

- ◆ `EvtWmiInstanceQueryInstance` — возвращает данные из указанного экземпляра WMI;
- ◆ `EvtWmiInstanceSetInstance` — записывает данные в указанный экземпляр WMI;
- ◆ `EvtWmiInstanceSetItem` — записывает значение одного элемента данных в экземпляре WMI;
- ◆ `EvtWmiInstanceExecuteMethod` — исполняет функцию, переданную ейзывающим клиентом и, необязательно, возвращает данные, полученные от этой функции. Такие методы обычно предназначены для динамического сбора данных.

KMDF активизирует эти функции обратного вызова в ответ на запросы от внешних пользователей WMI, которые указывают GUID поставщика WMI.

Образец драйвера Featured Toaster реализует функции обратного вызова для событий `EvtWmiInstanceQueryInstance`, `EvtWmiInstanceSetInstance` и `EvtWmiInstanceSetItem`. Большая часть кода в этих функциях обратного вызова предназначена для не специфичной для KMDF манипуляции данными.

### Пример: запрос данных у экземпляра WMI

Функция обратного вызова для события `EvtWmiInstanceQueryInstance` возвращает данные из указанного экземпляра WMI. В листинге 12.15 приведена реализация этой функции для обрата драйвера Toaster.

#### Листинг 12.15. Функция обратного вызова `EvtWmiInstanceQueryInstance` драйвера Toaster

```
NTSTATUS EvtWmiInstanceStdDeviceDataQueryInstance(
    IN     WDFWMINSTANCEx WmiInstance,
    IN     ULONG OutBufferSize,
    IN     PVOID OutBuffer,
    OUT    PULONG Buffer-Used)
```

```
{  
    PUCHAR          pBuf;  
    ULONG           size;  
    UNICODE_STRING string;  
    NTSTATUS        status;  
  
    PAGED_CODE();  
    string.Buffer = L"Aishwarya\0\0";  
    string.Length = (USHORT) (wcslen(string.Buffer) + 1) * sizeof(WCHAR);  
    string.MaximumLength = string.Length + sizeof(UNICODE_NULL);  
    size = ToasterDeviceInformation_SIZE + string.Length + sizeof(USHORT);  
    *BufferUsed = size;  
    if (OutBufferSize < size)  
    {  
        return STATUS_BUFFER_TOO_SMALL;  
    }  
    pBuf = (PUCHAR) OutBuffer;  
    // Копируем информацию структуры.  
    RtlCopyMemory(pBuf, ToasterWmiGetData(WmiInstance),  
                  ToasterDeviceInformation_SIZE);  
    pBuf += ToasterDeviceInformation_SIZE;  
    // Копируем строку. Помещаем длину строки перед самой строкой.  
    Status = WDF_WMI_BUFFER_APPEND_STRING(  
        pBuf,  
        size - ToasterDeviceInformation_SIZE,  
        &string,  
        &size);  
    return status;  
}
```

В качестве параметров функции обратного вызова передаются дескриптор экземпляра WMI, указатель на размер буфера вывода, в который драйвер будет записывать данные, указатель на сам буфер и указатель на область памяти, в которую драйвер сохранит количество записанных байтов.

В данном примере драйвер Featured Toaster возвращает данные из области контекста объекта экземпляра WMI вместе с произвольно образованной строкой, чтобы продемонстрировать обработку данных переменного размера. Драйвер записывает произвольно образованные данные в строку, вычисляет число байтов, требуемых для сохранения данных, и проверяет, поместятся ли они в предоставленный буфер вывода. Если размер буфера недостаточный, функция возвращает ошибку. В противном случае она устанавливает указатель на внутренний буфер, который отображается на буфер вывода, и с помощью метода `RtlCopyMemory` копирует содержимое области контекста объекта экземпляра WMI и длину этих данных в буфер.

Наконец драйвер вызывает функцию `WDF_WMI_BUFFER_APPEND_STRING`, чтобы добавить строку в буфер и откорректировать длину должным образом. Применение этой функции служит для обеспечения правильного выравнивания содержимого в блоке данных.

### Пример: организация экземпляра WMI

Функция обратного вызова для события `EvtWmiInstanceSetInstance` обновляет данные в указанном экземпляре WMI. В листинге 12.16 показан исходный код реализации этой функции в драйвере Featured Toaster.

**Листинг 12.16. Функция обратного вызова EvtWmiInstanceSetInstance драйвера Toaster**

```

NTSTATUS EvtWmiInstanceStdDeviceDataSetInstance(
    IN     WDFWMIINSTANCE WmiInstance,
    IN     ULONG InBufferSize,
    IN     PVOID InBuffer)
{
    if (InBufferSize < ToasterDeviceInformation_SIZE) {
        return STATUS_BUFFER_TOO_SMALL;
    }
    // Обновляем только перезаписываемые элементы.
    DebugLevel = ToasterWmiGetData(WmiInstance)>DebugPrintLevel =
        ((PToasterDeviceInformation) InBuffer)>DebugPrintLevel;
    return STATUS_SUCCESS;
}

```

В качестве параметров функции обратного вызова передаются дескриптор экземпляра WMI, указатель на размер буфера ввода, из которого драйвер считывает данные, и указатель на сам буфер.

Функция обновляет данные в области контекста. Первым делом она проверяет, достаточен ли размер буфера ввода, чтобы в него поместились структура ToasterDeviceInformation. Если размер буфера недостаточный, то функция возвращает ошибку. В противном случае функция обновляет структуру новыми данными из буфера, присваивая значение из буфера ввода соответствующему полю в области контекста. Записывать можно только в поле DebugPrintLevel, поэтому данный драйвер обновляет только это значение. Другие драйверы могут обновить весь блок или несколько записываемых полей. Функция возвращает статус успешного завершения.

**Пример: организация элемента данных WMI**

Функция обратного вызова *EvtWmiInstanceSetItem* обновляет определенный элемент в блоке данных WMI устройства. В листинге 12.17 показан исходный код реализации этой функции в драйвере Featured Toaster.

**Листинг 12.17. Функция обратного вызова EvtWmiInstanceSetItem драйвера Featured Toaster**

```

NTSTATUS EvtWmiInstanceStdDeviceDataSetItem(
    IN WDFWMIINSTANCE WmiInstance,
    IN ULONG DataItemId,
    IN ULONG InBufferSize,
    IN PVOID InBuffer)
{
    if (DataItemId == ToasterDeviceInformation_DebugPrintLevel_ID) {
        if (InBufferSize < sizeof(ULONG)) {
            return STATUS_BUFFER_TOO_SMALL;
        }
        DebugLevel = ToasterWmiGetData(WmiInstance)->DebugPrintLevel =
            *((PULONG) InBuffer);
        return STATUS_SUCCESS;
    }
}

```

```
    else {
        return STATUS_WMI_READ_ONLY;
    }
}
```

В качестве параметров этой функции обратного вызова передаются дескриптор экземпляра WMI, значение, идентифицирующее элемент данных, который нужно обновить, указатель на размер буфера ввода, из которого драйвер считывает новые данные, и указатель на сам буфер ввода.

В данном примерезывающий клиент может обновить только поле `DebugPrintLevel`, поэтому драйвер выполняет проверку на совпадение `DataItemId` с ID этого поля. Если идентификаторы совпадают, драйвер проверяет размер буфера точно так же, как и в функции обратного вызова `EvtWmiInstanceSetInstance`. Если размер буфера недостаточный, то драйвер возвращает ошибку.

В противном случае драйвер `Featured Toaster` обновляет данные элемента, таким же образом, как он обновляет данные экземпляра, и функция возвращает статус успешного завершения.

# ГЛАВА 13

## Шаблон UMDF-драйвера

Образец драйвера *Skeleton* представляет собой минимальный, но полностью функциональный драйвер UMDF. Код в этом образце драйвера поддерживает основные требования, которые должны реализовывать все драйверы UMDF, и для большинства драйверов этот код можно применять с незначительной модификацией или же вообще без какой-либо модификации. Благодаря этому, образец драйвера *Skeleton* не только служит хорошим примером создания кода в UMDF, но также может быть использован в качестве шаблона для реализации полнофункционального драйвера. Разработчики могут начать с базового рабочего драйвера и пошагово добавлять функциональность для поддержки их устройства, пока не получат драйвер, полностью удовлетворяющий их требованиям.

Такое использование образца драйвера *Skeleton* в качестве основы для реализации полнофункционального драйвера и рассматривается в этой главе.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Драйвер <i>Skeleton</i>	%wdk%\src\umdf\skeleton
<b>Документация WDK</b>	
UMDF Driver Skeleton Sample <sup>1</sup>	%wdk%\src\umdf\skeleton\skeleton.htm

### Описание образца драйвера *Skeleton*

Образец драйвера *Skeleton* поддерживает следующую функциональность, которую должны реализовывать все драйверы UMDF:

- ◆ инфраструктура DLL;
- ◆ базовая поддержка технологии COM;
- ◆ базовые реализации драйвера и объектов обратного вызова для устройства.

---

<sup>1</sup> Образец драйвера UMDF *Skeleton*. — *Пер.*

## Образец драйвера **Skeleton**

Образец драйвера **Skeleton** предоставляет файлы с кодом и следующие связанные с проектом файлы, которые можно использовать в качестве шаблона для организации проекта по разработке драйвера UMDF:

- ◆ набор вспомогательных файлов, используемых утилитой WDK Build для компоновки драйвера;
- ◆ INX-файл, который утилита Build преобразовывает в INF-файл для установки драйвера.

Данная функциональность является достаточной, чтобы получить полностью рабочий драйвер, но возможности этого драйвера очень ограничены. Образец драйвера **Skeleton** устанавливается и загружается, но для обработки большинства запросов ввода/вывода он зависит от стандартных возможностей, предоставляемых инфраструктурой.

В этой главе основное внимание уделяется вопросу адаптации кода образца драйвера **Skeleton** для создания полнофункционального драйвера UMDF. Образец драйвера **Skeleton** предоставляет надежную и функциональную модель для реализации остального кода драйвера. Если вы хотите применить другой подход к некоторым вопросам, рассматриваемых в этой главе, у вас не должно возникнуть никаких трудностей с модифицированием модели драйвера **Skeleton** под ваши нужды.

Способы реализации различных объектов и методов и причины их реализации в этой главе не рассматриваются.

Такие вопросы, как фабрики классов и интерфейс `IUnknown`, рассматриваются более подробно в главе 18.

### Примечание

Примеры, использованные в этой главе, были взяты из образца драйвера **Skeleton**. Большинство этих примеров были отредактированы для ясности и краткости. Для просмотра полного кода обратитесь к оригинальным материалам драйвера.

## Файлы образца драйвера **Skeleton**

Файлы образца драйвера **Skeleton** не только содержат код, но также демонстрируют оптимальные методики организации проекта разработки драйвера UMDF. Чтобы начать создавать драйвер, можно просто скопировать файлы образца драйвера в папку проекта. Эти файлы — модифицированные должным образом — можно использовать для базовой реализации, после чего можно добавлять по мере необходимости файлы для поддержки дополнительных возможностей, таких как, например, очереди запросов ввода/вывода.

### Исходные файлы

Большинство исходных файлов идут попарно: СРР-файл исходного кода и соответствующий заголовочный H-файл. Имена файлам присвоены такие, которые наиболее точно отображают их назначение. Обратите внимание на то, что код для некоторых очень простых методов, таких как, например, `IUnknown::AddRef`, может находиться в заголовочном файле. В следующих файлах содержится основная инфраструктура, и их, как правило, модифицировать нет надобности:

- ◆ `Dllsup.cpp` — реализует базовую инфраструктуру DLL;
- ◆ `Comsup.cpp` и `Comsup.h` — реализуют базовую поддержку COM: фабрику класса для объекта обратного вызова драйвера и базовую реализацию интерфейса `IUnknown`.

Следующие файлы содержат код, который необходимо модифицировать для поддержки определенного устройства:

- ◆ Driver.cpp и Driver.h — реализуют базовый объект обратного вызова для драйвера;
- ◆ Device.cpp и Device.h — реализуют базовый объект обратного вызова для устройства;
- ◆ Internal.h — содержит глобальные для проекта операторы #define и #include.

## **Файлы поддержки компоновки**

Следующие файлы содержат директивы и данные, используемые утилитой Build для компоновки DLL-файлов драйвера и родственных файлов:

- ◆ файл Sources — содержит список исходных файлов и родственную информацию;
- ◆ файлы Make — файлы Makefile и Makefile.inc содержат директивы для компоновки проекта;
- ◆ файл Exports — файл Exports.def указывает функции, которые DLL экспортирует по имени;
- ◆ файл версии ресурсов — файл Skeleton.rc содержит номер версии драйвера и родственную информацию.

## **Файлы для поддержки инсталляции**

INX-файлы, в сущности, являются независимыми от архитектуры версиями INF-файла. Утилита Build использует INX-файл для создания INF-файла для определенной сборки драйвера. Сборка INF-файлов из файлов INX является рекомендуемой практикой, поэтому при рассмотрении материала в этой главе предполагается, что в вашем проекте будет применяться INX-файл. Образец драйвера Skeleton содержит два следующих INX-файла:

- ◆ UMDFSkeleton\_OSR.inx — инструмент Stampinf использует этот INX-файл для создания INF-файла, который устанавливает образец драйвера Skeleton для обучающего устройства OSR USB Fx2.
- ◆ UMDFSkeleton\_Root.inx — инструмент Stampinf использует этот INX-файл для создания INF-файла, который устанавливает образец драйвера Skeleton как драйвер RED.

Применяйте тот INX-файл, который отвечает требованиям вашего проекта наилучшим образом. Файл UMDFSkeleton\_OSR.inx, возможно, является наиболее полезным файлом для большинства проектов, т. к. он устанавливает образец драйвера Skeleton в качестве драйвера устройства.

Подробное рассмотрение INX-файлов приводится в *главе 20*.

## **Модификация файлов образца драйвера Skeleton под собственные требования**

Исходные файлы образца драйвера Skeleton служат в качестве полезной отправной точки для разработки большинства драйверов UMDF. Большинство из них не требует никакой или очень незначительной модификации.

## Инфраструктура DLL

Инфраструктура DLL для драйвера UMDF обычно состоит из двух экспортруемых по имени функций: `DllMain` и `DllGetClassObject`. Обе функции реализованы в файле `Dllsup.cpp`, заголовочным файлом для которых является файл `Internal.h`.

### Функция `DllMain`

Функция `DllMain` является точкой входа DLL. Windows вызывает функцию `DllMain` после загрузки двоичного файла драйвера в хост-процесс и потом опять перед его выгрузкой. На действия, которые можно выполнять в `DllMain`, накладывается несколько ограничений, поэтому ее реализация обычно довольно проста.

Дополнительную информацию по функции `DllMain` см. в справке на нее в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=80069>.

Функция `DllMain`, в том виде, в каком она реализована в образце драйвера `Skeleton`, просто регистрирует и разрегистрирует трассировку WPP, как показано в листинге 13.1. Если в вашем драйвере применяется трассировка WPP, для него можно применить код образца драйвера `Skeleton`. Но при этом следует заменить определение `MYDRIVER_TRACING_ID` в заголовочном файле `Internal.h` идентификатором, уникальным для вашего драйвера. В функцию `DllMain` можно также добавить код для выполнения таких действий, как инициализация или освобождение глобальных переменных, если драйверу требуются эти операции.

#### Листинг 13.1. Реализация функции `DllMain` в образце драйвера `Skeleton`

```
BOOL WINAPI DllMain(HINSTANCE ModuleHandle,
                      DWORD Reason,
                      PVOID)
{
    if (DLL_PROCESS_ATTACH == Reason) {
        WPP_INIT_TRACING (MYDRIVER_TRACING_ID);
        // TODO: инициализировать глобальные переменные.
    }
    else if (DLL_PROCESS_DETACH == Reason) {
        WPP_CLEANUP();
        // TODO: освободить глобальные переменные.
    }
    return TRUE;
}
```

Если драйвер реализует трассировку WPP, необходимо также модифицировать заголовочный файл `Internal.h`, как показано в листинге 13.2 для образца драйвера `Skeleton`. Объяснение пронумерованных компонентов дается после листинга.

#### Листинг 13.2. Файл `Internal.h`, модифицированный для реализации трассировки WPP

```
//[1]
#define WPP_CONTROL_GUIDS \
WPP_DEFINE_CONTROL_GUID( \
    MyDriverTraceControl, \
    {e7541cdd,30e8,4b50,aeb0,51927330ae64}, \
    WPP_DEFINE_BIT(MYDRIVER_ALL_INFO) )
```

```

// [2]
#define WPP_FLAG_LEVEL_LOGGER(flag, level)      \
    WPP_LEVEL_LOGGER(flag)
#define WPP_FLAG_LEVEL_ENABLED(flag, level)      \
    (WPP_LEVEL_ENABLED(flag) &&               \
     WPP_CONTROL(WPP_BIT_## flag).Level >= level)

// begin_wpp config
// FUNC Trace{FLAG=MYDRIVER_ALL_INFO}(LEVEL, MSC, ...);
// end_wpp

// [3]
#define MYDRIVER_TRACING_ID L"Microsoft\\UMDF\\Skeleton"

```

Объяснение пронумерованных фрагментов листинга 13.2:

1. Замените значения в макросе `WPP_CONTROL_GUIDS` значениями, требующимися вашему драйверу.
  2. Образец драйвера Skeleton реализует одну функцию специального сообщения трассировки — `Trace`.
- Если вы хотите использовать другую функцию сообщения трассировки, выполните ее определение здесь.
3. Замените строку `"Microsoft\\UMDF\\Skeleton"` соответствующей строкой для вашего драйвера.

Тема трассировки WPP более подробно рассматривается в [главе 11](#).

### Функция `DllGetClassObject`

Загрузив DLL, инфраструктура вызывает функцию `DllGetClassObject`, чтобы получить указатель на любую из фабрик классов в DLL. Клиент может потом использовать это фабрику классов для создания экземпляра ассоциированного объекта COM. DLL в UMDF обычно имеют только одну фабрику классов — для объекта обратного вызова драйвера — поэтому обычно реализация функции `DllGetClassObject` не требует много кода. Большинство драйверов UMDF могут использовать функцию `DllGetClassObject` в том виде, в каком она реализована в образце драйвера Skeleton, без каких-либо дополнительных модификаций, если они также реализуют фабрику классов объекта обратного вызова драйвера как класс `CClassFactory`. В противном случае следует заменить `CClassFactory` именем соответствующего класса. Реализация функции `DllGetClassObject` в образце драйвера Skeleton показана в листинге 13.3.

#### Листинг 13.3. Реализация функции `DllGetClassObject` в образце драйвера Skeleton

```

_control_entrypoint(DllExport)
HRESULT STDMETHODCALLTYPE DllGetClassObject(
    _in REFCLSID ClassId,
    _in REFIID InterfaceId,
    _deref_out LPVOID *Interface)
{
    PCClassFactory factory;
    HRESULT hr = S_OK;

```

```
factory = new CClassFactory();
... // Код опущен.
return hr;
}
```

## Базовая поддержка технологии COM

Базовая поддержка технологии COM в образце драйвера Skeleton предоставляется двумя классами: классом `CUnknown`, который является базовой реализацией интерфейса `IUnknown`, и классом `CClassFactory`, который реализует фабрику классов для объекта обратного вызова драйвера. Код для этих двух классов расположен в файлах `Comsup.cpp` и `Comsup.h`.

### Класс `CUnknown`

Все объекты COM образца драйвера Skeleton реализуют интерфейс `IUnknown` путем наследования от класса `CUnknown`, впоследствии реализуя собственные, специфичные для интерфейса, версии трех методов интерфейса `IUnknown`. При условии, что вы придерживаетесь этой модели, класс `CUnknown`, в том виде как он реализован в образце драйвера Skeleton, можно применять без каких-либо дополнительных модификаций.

### Класс `CClassFactory`

Этот класс предназначен для создания нового экземпляра объекта обратного вызова драйвера. Класс `CClassFactory` предоставляет один интерфейс: `IClassFactory`, который поддерживает методы `CreateInstance` и `LockServer`.

Метод `CreateInstance` можно применять без каких-либо дополнительных модификаций в том виде, в каком он реализован в образце драйвера Skeleton, при соблюдении следующих условий:

- ◆ классу, реализующему объект обратного вызова драйвера, присваивается имя `CMyDriver`;
- ◆ `CMyDriver` реализуется таким же образом, как и в образце драйвера Skeleton — реализуя публичный сервисный метод `CreateInstance`, который создает экземпляр объекта обратного вызова драйвера.

В противном случае необходимо модифицировать имя класса объекта обратного вызова драйвера и код, занимающийся созданием экземпляра объекта. Реализация метода `CreateInstance` в образце драйвера Skeleton показана в листинге 13.4.

#### Листинг 13.4. Реализация метода `IClassFactory::CreateInstance` в образце драйвера Skeleton

```
HRESULT STDMETHODCALLTYPE CClassFactory::CreateInstance(
    _in_opt IUnknown * /* Outer-Object */,
    _in REFIID InterfaceId,
    _out PVOID *Object)
{
    HRESULT hr;
    PCMyDriver driver;
    *Object = NULL;
    hr = CMyDriver::CreateInstance(&driver);
    if (SUCCEEDED(hr))
    {
        hr = driver->QueryInterface(InterfaceId, Object);
```

```

    driver->Release();
}
return hr;
}

```

UMDF не вызывает метод LockServer фабрики классов, поэтому для этого метода требуется лишь символическая реализация, чтобы удовлетворить требование COM, что все методы интерфейса должны быть реализованы. Большинство драйверов UMDF могут просто использовать этот метод в том виде, в каком он реализован в образце драйвера Skeleton, без каких-либо дополнительных модификаций.

## Объект обратного вызова для образца драйвера Skeleton

Объект обратного вызова для образца драйвера Skeleton реализован в классе под названием CMYDriver, исходный код для которого находится в файлах Driver.cpp и Driver.h. Этот класс состоит из следующих трех основных компонентов:

- ◆ публичного вспомогательного метода CreateInstance, который создает экземпляр объекта обратного вызова драйвера;
- ◆ специфической для интерфейса реализации IUnknown;
- ◆ реализации интерфейса IDriverEntry.

### Метод *CreateInstance*

Эта реализация метода CreateInstance не является частью интерфейса IClassFactory, но он выполняет ту же роль, предоставляя для других классов в DLL удобный механизм для создания экземпляра класса CMYDriver. В данном случае фабрика классов объекта обратного вызова драйвера вызывает метод CreateInstance. Другие классы образца драйвера Skeleton реализуют подобные методы.

Код метода CMYDriver::CreateInstance для создания объекта можно использовать без модификаций, но, возможно, понадобится добавить код для инициализации объекта должным образом. В образце драйвера Skeleton код инициализации размещен во вспомогательном закрытом (private) методе CMYDriver::Initialize, который вызывается методом CreateInstance. Как показано в листинге 13.5, в образце драйвера Skeleton метод Initialize реализуется в виде заполнителя, который просто возвращает S\_OK. Поэтому для своего драйвера вам нужно будет добавить к методу необходимый код для выполнения инициализации.

#### Листинг 13.5. Реализация метода CMYDriver::CreateInstance в образце драйвера Skeleton

```

HRESULT CMYDriver::CreateInstance(_out PCMyDriver *Driver)
{
    PCMyDriver driver; HRESULT hr;
    driver = new CMYDriver();
    ... // Код опущен.
    hr = driver->Initialize();
    if (SUCCEEDED(hr))
    {
        *Driver = driver;
    }
}

```

```
... // Код опущен.  
return hr;  
}  
HRESULT CMYDriver::Initialize(VOID)  
{  
    //TODO: Добавить код для инициализации.  
    return S_OK;  
}
```

## Интерфейс *IUnknown*

Класс CMYDriver наследует от интерфейса CUnknown, который является базовой реализацией интерфейса IUnknown, как было рассмотрено в разд. "Базовая поддержка технологии COM" ранее в данной главе. В этой главе предполагается, что для реализации интерфейса IUnknown вы будете применять эту модель. В таком случае, вы можете применить методы интерфейса IUnknown для объекта обратного вызова драйвера в том виде, в каком они реализованы в образце драйвера Skeleton, без каких-либо модификаций.

## Интерфейс *IDriverEntry*

Основная функциональность класса CMYDriver реализована в интерфейсе IDriverEntry, который поддерживает методы OnInitialize, OnDeinitialize и OnDeviceAdd.

- ◆ Метод OnInitialize вызывается при загрузке драйвера, чтобы инициализировать его, а метод OnDeinitialize вызывается перед выгрузкой драйвера, чтобы позволить драйверу выполнить необходимую очистку.

Образец драйвера Skeleton не выполняет никакой инициализации или deinициализации, поэтому методы OnInitialize и OnDeinitialize реализованы в нем только формально. Если ваш драйвер должен выполнять какую-либо инициализацию при загрузке или очистку перед выгрузкой, в эти методы необходимо добавить соответствующий код.

- ◆ Метод OnDeviceAdd является ключевым методом интерфейса. Его главное назначение — создание объекта устройства и добавление его в стек.

Он также применяется для выполнения инициализации устройства и изменения настроек, специфичных для устройства. В большинстве драйверов этот метод в том виде, в каком он реализован в образце драйвера Skeleton, применять нельзя; для этого над ним необходимо выполнить несколько модификаций.

В листинге 13.6 приводится исходный код для инициализации образца драйвера Skeleton, выполняемой методом IDriverEntry. Объяснение пронумерованных комментариевдается после листинга.

### Листинг 13.6. Реализация метода sample IDriverEntry::OnDeviceAdd в образце драйвера Skeleton

```
HRESULT CMYDriver::OnDeviceAdd(  
    _in IWDFDriver *FxWdfDriver,  
    _in IWDFDeviceInitialize *FxDeviceInit)  
{  
    HRESULT hr;  
    PCMyDevice device = NULL;  
    // [1] TODO: Выполнить всю инициализацию для каждого устройства.
```

```

// [2] Создается объект устройства.
hr = CMYDevice::CreateInstance(FxWdfDriver, FxDeviceInit, &device);
// [3] TODO: изменить настройки для каждого устройства.
// [4] Вызвать метод конфигурирования объекта
// обратного вызова устройства.
if (SUCCEEDED(hr))
{
    hr = device->Configure();
}
... // Код опущен.
}

```

Объяснение пронумерованных фрагментов листинга 13.6.

1. Прежде чем создавать объект устройства, необходимо выполнить всю требуемую инициализацию индивидуально для каждого устройства.
2. Немодифицированный код образца драйвера Skeleton можно использовать для создания объекта устройства при условии, что объект обратного вызова устройства реализуется так же, как и в образце драйвера Skeleton. А именно:
  - классу, реализующему объект обратного вызова устройства, присваивается имя CMyDevice;
  - этот класс имеет публичный метод CreateInstance, который создает объект обратного вызова устройства и экземпляр объекта устройства.

Оба метода рассматриваются более подробно далее в этой главе.

3. Прежде чем завершить инициализацию, конфигурируя устройство, выполняются изменения любых настроек, предоставляемых объектом, индивидуально для каждого устройства.
4. Выполняется конфигурация устройства вызовом метода CMYDevice::Configure.

Этот метод завершает процесс инициализации, выполняя такие задачи, как создание объектов очередей для управления запросами ввода/вывода. Другая стандартная задача, которую выполняет этот метод, — это создание и включение интерфейса драйвера.

Если ваш объект обратного вызова устройства применяет другой подход к конфигурации, то вам необходимо модифицировать соответствующий код должным образом. Метод Configure обсуждается далее в этой главе.

Оставшийся код метода OnDeviceAdd в листинге не приводится, т. к. в нем просто выполняется очистка перед возвращением управления и его можно применить в большинстве драйверов без каких-либо предварительных модификаций. Для получения подробностей см. исходный код образца драйвера.

## Необязательные интерфейсы

Объект обратного вызова драйвера предоставляет только интерфейсы IUnknown и IDriverEntry, поэтому для него не реализуются никакие необязательные интерфейсы.

## Объект обратного вызова устройства в образце драйвера Skeleton

Объект обратного вызова устройства в образце драйвера Skeleton обычно требует значительной модификации. Образец драйвера Skeleton не поддерживает никакого настоящего

устройства, поэтому в нем реализованы очень немногие возможности, требуемые драйверами устройств. Объект обратного вызова устройства реализован в классе под именем `CMyDevice`, исходный код которого находится в файлах `Device.cpp` и `Device.h`.

## Вспомогательные методы для объекта обратного вызова устройства

Объект обратного вызова устройства реализует вспомогательные методы `CreateInstance`, `Initialize` и `Configure`.

### Метод `CreateInstance`

Этот открытый (public) вспомогательный метод создает экземпляр класса `CMyDevice`, после чего вызывает метод `Initialize`, который создает объект устройства UMDF. Большинство драйверов UMDF может просто использовать этот метод в том виде, в каком он реализован в образце драйвера `Skeleton` без каких-либо дополнительных модификаций.

### Метод `Initialize`

Этот публичный вспомогательный метод выполняет несколько важных задач. Для большинства драйверов этот метод (листинг 13.7) требует значительных модификаций, как изложено в примечаниях после листинга.

#### Листинг 13.7. Реализация метода `CMyDevice::Initialize` в образце драйвера `Skeleton`

```
HRESULT CMyDevice::Initialize(
    _in IWDFDriver           * FxDriver,
    _in IWDFDeviceInitialize * FxDeviceInit)
{
    IWDFDevice *fxDevice;
    HRESULT hr;
    // [1] TODO: установить область блокировки
    FxDeviceInit->SetLockingConstraint(WdfDeviceLevel);
    // [2] TODO: драйвер фильтра должен указать это
    //        обстоятельство здесь.
    // FxDeviceInit->SetFilter();
    // [3] TODO: выполнить вся требуемую инициализацию
    //        для каждого устройства.
    ... // Код опущен.
    // [4] Создать объект устройства. Код опущен.
}
```

Объяснение пронумерованных фрагментов листинга 13.7.

1. Установите область блокировки для драйвера, вызвав метод `IWDFDeviceInitialize::SetLockingConstraint`.

В образце драйвера `Skeleton` применяется область блокировки `WdfDeviceLevel`, поэтому, если в вашем драйвере применяется иная область, это значение нужно изменить.

2. Если вы реализуете драйвер фильтра, необходимо вызвать метод `IWDFDeviceInitialize::SetFilter`. Это можно сделать, просто раскомментировав соответствующую строку кода в образце драйвера `Skeleton`.
3. Здесь добавляется код для любой требуемой для каждого устройства инициализации.

4. Для создания объекта устройства в большинстве драйверов можно использовать немодифицированный каким-либо образом код из образца драйвера Skeleton, поэтому этот код опущен в листинге 13.7.

### **Метод *Configure***

Как рассматривалось ранее, объект обратного вызова драйвера вызывает метод *Configure* объекта обратного вызова устройства после создания объекта устройства UMDF. Основными задачами метода *Configure* являются следующие:

- ◆ создание и разрешение интерфейса устройства;
- ◆ создание и конфигурирование объектов очередей ввода/вывода для управления входящими запросами ввода/вывода.

Так как образец драйвера Skeleton не обрабатывает никаких запросов ввода/вывода, то выполнена лишь символическая реализация метода *Configure*, который только возвращает `s_ok`. В этот метод можно добавить требуемый вашему драйверу код. Для примера реализации метода *Configure* см. исходный код образца драйвера `Fx2_Driver`.

### **Интерфейс *IUnknown***

Если интерфейс `IUnknown` реализуется на основе образца драйвера Skeleton, то для реализации специфичных для интерфейса методов интерфейса `IUnknown` можно использовать немодифицированный код драйвера Skeleton.

### **Необязательные интерфейсы**

Для объекта обратного вызова устройства не является обязательным иметь какие-либо интерфейсы. Но т. к. большинство устройств поддерживает Plug and Play, то они должны предоставлять интерфейс `IPnpCallback` и, возможно, интерфейсы `IPnpCallbackHardware` и `IPnpCallbackSelfManagedIo`. Так как драйвер Skeleton не управляет каким-либо устройством, то он не реализует никакой из этих интерфейсов. Для драйверов, поддерживающих Plug and Play, самым простым подходом будет добавить код, необходимый для требуемых интерфейсов, в класс `CMyDevice`.

Для примера, каким образом это делать, см. образец драйвера `Fx2_Driver`.

### **Что дальше?**

Любые последующие модификации образца драйвера Skeleton зависят от требований конкретного драйвера. Наиболее распространенной модификацией является реализация одного или нескольких объектов обратного вызова очереди для получения запросов ввода/вывода из ассоциированных очередей. Широко применяемой практикой является реализация каждого объекта обратного вызова очереди в виде отдельного класса в отдельном файле.

В качестве отправной точки можно использовать образец драйвера Echo, который реализует один объект обратного вызова для обработки запросов на чтение, запись и IOCTL. Данный объект обратного вызова очереди реализован в виде одного класса и находится в файле `Queue.cpp`.

В образце драйвера применяется более продвинутый подход, при котором запросы на чтение и запись обрабатываются одним объектом обратного вызова очереди, а запросы IOCTL — другим. Оба класса, реализующие эти два объекта наследуют от родительского класса, который реализует общую функциональность, требуемую всеми объектами обратного вызова очереди.

# Модификация вспомогательных файлов образца драйвера Skeleton

При компоновке проекта драйвера утилита Build пользуется вспомогательными файлами для компоновки. Для большинства драйверов UMDF можно просто скопировать вспомогательные файлы образца драйвера Skeleton и модифицировать их, как требуется для нужд драйвера.

Дополнительная информация о вспомогательных файлах представлена в *главах 19 и 20*.

## Файл Sources

Файл Sources содержит список исходных файлов и родственную информацию, требуемую утилитой Build. Этой файл необходимо модифицировать в нескольких местах, как показано в листинге 13.8.

### Листинг 13.8. Файл Sources образца драйвера Skeleton

```
# [1]
UMDF_VERSION=1
UMDF_MINOR_VERSION=5
#[2]
TARGETNAME=UMDFSkeleton
TARGETTYPE=DYNLINK
USE_MSVCRT=1
C_DEFINES = $(C_DEFINES) /D_UNICODE /DUNICODE
WIN32_WINNT_VERSION=$(LATEST_WIN32_WINNT_VERSION)
_NT_TARGET_VERSION=$(NT_TARGET_VERSION_WINXP)
NTDDI_VERSION=$(LATEST_NTDDI_VERSION)
DLLENTRY=_Dll MainCRTStartup
DLLDEF=exports.def
#[3]
SOURCES=\
    Skeleton.re          \
    dllsup.cpp          \
    comsup.cpp          \
    driver.cpp          \
    device.cpp
#[4]
TARGETLIBS=\
    $(SDK_LIB_PATH)\strsafe.lib      \
    $(SDK_LIB_PATH)\kernel32.lib     \
    $(SDK_LIB_PATH)\advapi 32.lib
#[5]
NTTARGETFILES=$(OBJ_PATH)\$(0)\UMDFSkeleton_Root.inf\
               $(OBJ_PATH)\$(0)\UMDFSkeleton_OSR.inf
MISCFILES=$(NTTARGETFILES)
#[6]
RUN_WPP= $(SOURCES) -dll -scan:internal.h
```

Объяснение пронумерованных фрагментов листинга 13.8.

1. Данная версия образца драйвера была реализована для UMDF версии 1.5. Для более поздних версий UMDF необходимо заменить номер версии на соответствующий.

2. Переменная TARGETNAME указывает имена нескольких выходных файлов, включая двоичный файл драйвера. Измените это имя соответствующим образом для вашего драйвера.
3. В операторе SOURCES указываются исходные файлы, которые нужно скомпилировать. Модифицируйте имена файлов и добавьте файлы, как требуется для вашего проекта.
4. В операторе TARGETLIBS указываются имена файлов статических библиотек, связываемых с драйвером. Модифицируйте этот список должным образом для своего проекта.
5. В операторе NTTARGETFILES указывается, что проект включает файл Makefile.inc.

Файл Makefile.inc для образца драйвера Skeleton содержит инструкции для создания INF-файла из файла INX проекта. Значение, присвоенное NTTARGETFILES, указывает имя INF-файла. Измените его на подходящее значение для своего драйвера.

### Примечание

Образец драйвера производит два INF-файла: один для установки драйвера в качестве драйвера RED, а другой для установки драйвера Skeleton в качестве драйвера для устройства Fx2. Для большинства проектов требуется только один INF-файл, чтобы установить драйвер для ассоциированного устройства.

6. Оператор RUN\_WPP запускает препроцессор WPP. В последнем аргументе этого оператора указывается, что образец драйвера Skeleton имеет функцию специального извещения трассировки, которая определена в файле Internal.h. Если специальное сообщение трассировки не используется или если функция определена по-другому, этот аргумент необходимо удалить или модифицировать.

## Файлы Make

Файлы Make — Makefile и Makefile.inc содержат директивы для компоновки проекта. Они используются следующим образом.

- ◆ Файл Makefile должен быть одинаковым для всех проектов драйвера.  
Файл Makefile из образца драйвера Skeleton можно использовать без каких-либо дополнительных модификаций.
- ◆ Файл Makefile.inc содержит инструкции для создания INF-файла из INX-файла проекта. Этот файл также можно использовать без каких-либо дополнительных модификаций.  
Но если в проекте необходимо создать дополнительные специальные INF-файлы, добавьте необходимые директивы в файл Makefile.inc.

## Файл Exports

В файле Exports.def (листинг 13.9) указываются функции, которые нужно экспорттировать по имени из DLL. Образец драйвера Skeleton экспорттирует одну функцию — DllGetClassObject. Это типично для драйверов UMDF, поэтому в большинстве проектов этот файл, в том виде, в каком он представлен в драйвере Skeleton, можно использовать с одной модификацией: необходимо заменить имя UMDFSkeleton.DLL именем вашего двоичного файла.

### Листинг 13.9. Файл Exports.def образца драйвера Skeleton

```
LIBRARY "UMDFSkeleton.DLL"
EXPORTS
    DllGetClassObjectPRIVATE
```

## Файл версии ресурсов

Файл Skeleton.rc содержит информацию для идентификации драйвера и о его версии. Для своего драйвера измените имя файла и строки, выделенные жирным шрифтом, соответствующим образом.

### Листинг 13.10. Файл версии для образца драйвера Skeleton

```
#include <windows.h>
#include <ntverp.h>
#define VER_FILETYPE VFT_DLL
#define VER_FILESUBTYPE VFT_UNKNOWN
#define VERFILEDESCRIPTION_STR
    "WDF:UMDF Skeleton User-Mode Driver Sample"
#define VER_INTERNALNAME_STR      "UMDFSkeleton"
#define VER_ORIGINALFILENAME_STR  "UMDFSkeleton.dll"
#include "common.ver"
```

## Файл INX

Для создания INF-файла из файла INX утилита Build вызывает инструмент Stampinf.

Файл INX почти идентичен соответствующему файлу INF. Разница состоит в том, что файл INX содержит несколько маркеров, которые инструмент Stampinf заменяет соответствующими значениями для сборки:

- ◆ маркер \$ARCH\$ заменяется значением процессорной архитектуры, для которой компилируется драйвер;
- ◆ маркер \$UMDFVERSION\$ заменяется правильной версией UMDF;
- ◆ маркер \$UMDFCOINSTALLERVERSION\$ заменяется правильной версией соинсталлятора UMDF.

Чтобы использовать файл INX из образца драйвера Skeleton в качестве основы для установки вашего собственного драйвера, необходимо модифицировать несколько установок.

Дополнительная информация о файлах INX и INF приводится в главе 20.

В листинге 13.11 показан пример из файла UMDFSkeleton\_OSR.inx, который устанавливает образец драйвера Skeleton в качестве драйвера для устройства Fx2. Файл UMDFSkeleton\_Root.inx здесь не рассматривается, т. к. относительно небольшое число драйверов устанавливается в качестве драйверов RED. Файл UMDFSkeleton\_Root.inx похож на файл UMDFSkeleton\_OSR.inx, но в нем отсутствуют несколько директив и значений, требуемых драйверами устройств. Обратите внимание на то, что строки определяются в конце файла и вставляются в директивы в формате %StringName%.

### Листинг 13.11. Файл INX образца драйвера Skeleton

```
; [1]
[Version]
Signature="$Windows NT$"
Class=Sample
ClassGuid={78A1C341-4539-11d3-B88D-00C04FAD5171}
Provider=%MSFTUMDF%
DriverVer=03/25/2005,0.0.0.1
CatalogFile=wudf.cat
```

```
; [2]
[Manufacturer]
%MSFTUMDF%=Microsoft, NT$ARCH$


; [3]
[Microsoft.NT$ARCH$]
%SkeletonDeviceName%=Skeleton_Install, USB\VID_045e&PID_94aa&mi_00
%SkeletonDeviceName%=Skeleton_Install, USB\VID_0547&PID_1002


; [4]
[ClassInstall32]
AddReg=SampleClass_RegistryAdd
[SampleClass_RegistryAdd]
HKR,,,,%ClassName%
HKR,,Icon,-10


; [5]
[SourceDiskFiles]
UMDFSkeleton.dll=1
WudfUpdate_$UMDFCOINSTALLERVERSION$.dll=1
WdfCoInstaller01005.dll=1
WinUsbCoinstaller.dll=1


; [6]
[SourceDiskNames]
1 = %MediaDescription%


; [7]
[Skeleton_Install.NT]
CopyFiles=UMDriverCopy
Include=WINUSB.INF
Needs=WINUSB.NT


; [8]
[Skeleton_Install.NT.hw]
AddReg=Skeleton_Device_AddReg


; [9]
[Skeleton_Install.NT.Services]
AddService=WUDFRd,0x000001fa,WUDFRD_ServiceInstall
AddService=WinUsb,0x000001f8,WinUsb_ServiceInstall
[Skeleton_Install.NT.CoInstallers]
CopyFiles=CoInstallers_CopyFiles
AddReg=CoInstallers_AddReg


; [10]
[Skeleton_Install.NT.Wdf]
KmdfService = WINUSB, WinUsb_Install
UmdfService = UMDFSkeleton, UMDFSkeleton_Install
UmdfServiceOrder = UMDFSkeleton
[WinUsb_Install]
KmdfLibraryVersion = 1.5
```

```
[UMDFSkeleton_Install]
UmdfLibraryVersion=%UMDFVERSION%
DriverCLSID="{d4112073-d09b-458f-a5aa-35ef21eef5de}"
ServiceBinary="%12%\umdf\UMDFSkeleton.dll"

[Skeleton_Device_AddReg]
HKR,, "LowerFilters", 0x00010008, "WinUsb"

[WUDFRD_ServiceInstall]
DisplayName = %WudfRdDisplayName%
ServiceType = 1
StartType = 3
ErrorControl = 1
ServiceBinary = %12%\WUDFRd.sys
LoadOrderGroup = Base

; [11]
[WinUsb_ServiceInstall]
DisplayName = %WinUsb_SvcDesc%
ServiceType = 1
StartType = 3
ErrorControl = 1
ServiceBinary = %12%\WinUSB.sys

; [12]
[CoInstallers_AddReg]
HKR,, CoInstallers32, 0x00010000,
    "WudfUpdate_%UMDFCOINSTALLERVERSION%.dll",
"WinUsbCoInstaller.dll", "WdfCoInstaller01005.dn,WdfCoInstaller"
[CoInstallers_CopyFiles]
WudfUpdate_%UMDFCOINSTALLERVERSION%.dll
WdfCoInstaller01005.dll
WinUsbCoInstaller.dll
[DestinationDirs]
CoInstallers_CopyFiles=11 ; copy to system32
UMDriverCopy=12, UMDF ; copy to drivers/umdf

; [13]
[UMDriverCopy]
UMDFSkeleton.dll

; [14]
[Strings]
MSFTUMDF="Microsoft Internal (WDF:UMDF)"
MediaDescription="Microsoft Sample Driver Installation Media"
ClassName="WUDF Sample"
WudfRdDisplayName="Windows Driver Foundation -
                    User-mode Driver Framework Reflector"
SkeletonDeviceName="Microsoft Skeleton User-Mode
                    Driver on OSR USB Device Sample"
WinUsb_SvcDesc="WinUSB Driver"
```

Объяснение пронумерованных фрагментов листинга 13.11.

1. В разделе **[Version]** измените значения переменных **Class** и **ClassGuid** на значения, требуемые вашим устройством. Замените **%MSFTUMDF%** и ассоциированную строку на подходящее значение для своего драйвера.

Дополнительную информацию о значениях переменных Class и ClassGuid см. в разделе **Device Setup Classes** (Классы для установки устройства) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80620>.

2. В разделе [Manufacturer] замените значение %MSFTUMDF% новым значением. Строку "Microsoft" нужно заменить именем вашей компании.
3. В разделе [Microsoft.NT\$ARCH\$]:
  - замените строку "Microsoft" в имени раздела именем компании, указанном в разделе [Manufacturer];
  - замените значение %SkeletonDeviceName% соответствующей строкой для вашего драйвера. Эта строка задает значение DDInstall, которое используется дальше в файле. Замените все случаи "Skeleton\_Install" в дальнейшей части файла на указанную строку;
  - измените идентификаторы аппаратного обеспечения на значения для вашего устройства.
4. В разделе [ClassInstall32] замените %ClassName% в соответствующем разделе [SampleClass\_RegistryAdd] на требуемое значение для вашего устройства.
5. В разделе [SourceDiskFiles]:
  - замените имя UMDFSkeleton.DLL именем вашего двоичного файла;
  - файл WinUsbCoInstaller.dll является сонсталлятором для WinUSB и требуется только для драйверов USB;
  - файл WdfCoInstaller01005.dll является сонсталлятором KMDF. Он включен здесь, т. к. USB-драйверы UMDF зависят от механизма WinUSB, который основан на KMDF. При реализации драйвера USB или драйвера другого типа с компонентами, зависящими от KMDF, оставьте эту директиву, но, если необходимо, обновите значение для версии. В противном случае удалите эту директиву.
6. В разделе [SourceDiskNames] измените строку, присвоенную %MediaDescription%, на соответствующее значение для вашего драйвера.
7. Измените Skeleton\_Install в последующих разделах на значение DDInstall, указанное в шаге 3.

Директивы Include и Needs в разделе [DDInstall.NT] требуются для большинства драйверов. В противном случае удалите эти директивы.

8. В этом разделе и в соответствующем разделе [Skeleton\_Device\_AddReg] WinUSB устанавливается в качестве нижнего драйвера фильтра для образца драйвера Skeleton. Он требуется для драйверов USB, а также может быть использован драйверами других типов для таких целей, как установка драйвера фильтра.

Если реализуется драйвер USB:

- в разделе [Skeleton\_Install.NT.hw] замените Skeleton\_Device\_AddReg и соответствующее имя раздела на требуемое для драйвера значение;
  - используйте директиву HKR в разделе [Skeleton\_Device\_AddReg] без модификаций.
9. В разделе [DDInstall.NT.Services]:
    - если драйвер является драйвером фильтра, измените раздел [DDInstall.NT.Services], чтобы установить отражатель — WUDFRd — в качестве драйвера фильтра в стеке устройств режима ядра;

- если драйвер является функциональным драйвером, раздел [DDInstall.NT.Services] следует изменить, чтобы установить рефлектор в качестве сервиса для устройства;
- если драйвер не является драйвером USB, удалите вторую директиву AddService, которая добавляет сервис WinUSB.

### Примечание

Второй элемент значения, указываемого директивой AddService, является флаговым полем. В первом бите этого поля указывается, является ли драйвер драйвером фильтра или функциональным драйвером.

Дополнительную информацию по флаговым установкам см. в разделе **INF AddService Directive** (Директива AddService файла INF) в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=81348>.

#### 10. В разделе [DDInstall.Wdf]:

- при реализации драйвера USB используйте директиву KmdfService и ассоциированный раздел [WinUsb\_Install], но если необходимо, измените значение директивы KmdfLibraryVersion на значение соответствующей версии. При реализации не-USB-драйвера, зависимого от KMDF, необходимо модифицировать значение директивы KmdfService соответствующим образом. В противном случае удалите эту директиву и соответствующий раздел;
- измените значения директивы UmdfService соответственно вашему драйверу. Присвойте директиве UMDFServiceOrder имя драйвера;
- если драйвер выполняет имперсонацию, добавьте директиву UMDFImpersonation, которая указывает максимальный уровень имперсонации для драйвера;
- в подразделе [UMDFSkeleton\_Install] измените имя раздела на значение, указанное в директиве UmdfService. Замените значение ServiceBinary именем двоичного файла драйвера. Измените значение DriverCLSID значением CLSID объекта обратного вызова вашего драйвера.

#### 11. При реализации драйвера USB используйте раздел [WinUsb\_ServiceInstall] без модификаций. В противном случае удалите этот раздел.

#### 12. На разделы [CoInstallers\_AddReg] и [CoInstallers\_CopyFiles] ссылаются разделы [DDInstall.NT.Coinstallers]:

- при реализации драйвера, зависящего от KMDF, если необходимо, обновите номер версии для WdfCoInstaller01005.dll;
- при реализации не-USB-драйвера удалите значение WinUsbCoinstaller.dll;
- при реализации драйвера, не зависящего от KMDF, удалите значения WdfCoInstaller01005.dll и WdfCoInstaller.

#### 13. В разделе [UMDriverCopy] замените UMDFSkeleton.dll именем двоичного файла вашего драйвера.

#### 14. В разделе [Strings] измените директивы и значения соответствующими значениями для вашего драйвера.

В зависимости от типа устройства, поддерживаемого драйвером, и от того, является ли драйвер чисто программным, например, драйвером фильтра, может быть необходимым внести дополнительные изменения в файл INX.

# **ЧАСТЬ IV**

## **Смотрим глубже — больше о драйверах WDF**

**Глава 14.** За пределами инфраструктур

**Глава 15.** Планирование, контекст потока и уровни IRQL

**Глава 16.** Аппаратные ресурсы и прерывания

**Глава 17.** Прямой доступ к памяти

**Глава 18.** Введение в COM

## ГЛАВА 14

# За пределами инфраструктур

Хотя и UMDF, и KMDF предоставляют возможности, как правило, достаточные для большинства драйверов, драйверам иногда приходится выходить за пределы инфраструктур, чтобы воспользоваться сервисами, не предоставляемыми инфраструктурами, или же чтобы обработать запросы ввода/вывода, не поддерживаемые инфраструктурами. В этой главе описывается использование в драйверах WDF системных возможностей, не входящих в инфраструктуры.

Ресурсы, необходимые для данной главы	Местонахождение
<b>Образцы драйверов</b>	
Serial	%wdk%\Src\Kmdf\Serial
Toastmon	%wdk%\Src\Kmdf\Toaster\Toastmon
<b>Документация WDK</b>	
Kernel-Mode Driver Architecture <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79288">http://go.microsoft.com/fwlink/?LinkId=79288</a>

## Использование системных сервисов, не входящих в состав инфраструктур

Кроме сервисов, предоставляемых инфраструктурами, драйверы UMDF и KMDF могут использовать следующие системные сервисы:

- ◆ драйверы UMDF могут использовать многие функции Windows API. Основным исключением являются функции, которые создают и манипулируют пользовательскими интерфейсами;
- ◆ драйверы KMDF могут использовать системные функции режима ядра, включая функции, которые манипулируют объектами WDM, лежащими в основе объектов WDF.

---

<sup>1</sup> Архитектура драйверов режима ядра. — Пер.

## Использование функций Windows API в драйверах UMDF

С определенными ограничениями, драйверы UMDF могут пользоваться Windows API для исполнения задач, не поддерживаемых инфраструктурой. Например:

- ◆ драйвер, для которого требуется таймер или рабочий поток, получает доступ к этим возможностям с помощью Windows API;
- ◆ драйвер, требующий события для подачи сигнала между потоками или требующий блокировку иного типа, нежели блокировка для каждого объекта индивидуально, предоставляемая методом `wdfObject::AcquireLock`, может использовать событие, семафор или другой блокировочный примитив Windows;
- ◆ драйверу, отправляющему запросы ввода/вывода устройству, не поддерживающему объектами получателя ввода/вывода UMDF, может потребоваться вызывать функции Win32 или процедуры RPC (Remote Procedure Call, удаленный вызов процедуры), чтобы послать запрос ввода/вывода и обработать завершение;
- ◆ драйвер, который использует свойства своего устройства, не предоставляемые UMDF, может воспользоваться функциями семейства `SetupDiXXX`, чтобы получить эту информацию.

Драйверы UMDF вызывают функции Windows API таким же образом, как это делает любое другое приложение пользовательского режима. В листинге 14.1 приведен пример возможного использования драйвером UMDF функций семейства `SetupDiXXX`. Этот пример был адаптирован из образца драйвера `Fx2_Driver`.

### Листинг 14.1. Вызов функций Windows API в драйвере UMDF

```
ULONG instanceIdCch = 0;
PWSTR instanceId = NULL;
HDEVINFO deviceInfoSet = NULL;
SP_DEVINFO_DATA deviceInfo = {sizeof(SP_DEVINFO_DATA)};
HRESULT hr;
hr = m_FxDevice->RetrieveDeviceInstanceId(NULL, &instanceIdCch);
if (FAILED(hr)) {
    . . . // Код для обработки ошибки опущен.
}
instanceId = new WCHAR[instanceIdCch];
if (instanceId == NULL) {
    . . . // Код для обработки ошибки опущен.
}
hr = m_FxDevice->RetrieveDeviceInstanceId(instanceId, &instanceIdCch);
if (FAILED(hr)) {
    . . . // Код для обработки ошибки опущен.
}
deviceInfoSet = SetupDiCreateDeviceInfoList(NULL, NULL);
if (deviceInfoSet == INVALID_HANDLE_VALUE) {
    . . . // Код для обработки ошибки опущен.
}
if (SetupDiOpenDeviceInfo(deviceInfoSet,
                         instanceId,
                         NULL,
```

```
0,
&DeviceInfo) == FALSE) {
. . . // Код для обработки ошибки опущен.
}
if (SetupDiGetDeviceRegistryProperty(deviceInfoSet,
    &DeviceInfo,
    SPDRP_BUSTYPEGUID,
    NULL,
    (PBYTE) &m_BusTypeGuid,
    sizeof(m_BusTypeGuid),
    NULL) == FALSE) {
. . . // Код для обработки ошибки опущен.
}
SetupDiDestroyDeviceInfoList(deviceInfoSet);
delete[] instanceId;
```

Для образца драйвера, приведенного в листинге, требуется идентификатор GUID для типа шины, к которой подключено устройство, который нельзя получить в UMDF. Но если драйвер имеет идентификатор экземпляра устройства, он может получить этот идентификатор, вызывая функции семейства SetupDiXXX. В представленном листинге драйвер получает указатель на идентификатор экземпляра устройства, вызывая метод IWDFDevice::RetrieveDeviceInstanceId. После этого драйвер вызывает в последовательности несколько других функций семейства SetupDiXXX. Функция SetupDiCreateDeviceInfoList создает информационный набор устройства, который драйвер передает функции SetupDiOpenDeviceInfo вместе с идентификатором экземпляра устройства, чтобы получить информацию об устройстве. Наконец, драйвер вызывает функцию SetupDiGetDeviceRegistryProperty, чтобы извлечь идентификатор GUID типа шины из полученной информации об устройстве. После этого драйвер уничтожает информационный набор устройства и удаляет идентификатор экземпляра устройства.

При любой возможности драйверы должны использовать инфраструктурные интерфейсы для создания, отправления и завершения запросов ввода/вывода, для блокировки объектов инфраструктуры и для взаимодействия с системными компонентами, какими как, например, механизм WinUSB. Инфраструктурные интерфейсы предоставляют дополнительные возможности для контроля ошибок и проверки достоверности параметров, а также такие дополнительные, особенно полезные для драйверов возможности, как функции обратного вызова для завершения запросов.

Например, вместо использования функции Windows CreateFile, ReadFile и WriteFile для создания и отправления запросов ввода/вывода, драйверы UMDF должны вызывать методы IWDFDevice::CreateRequest и IWdfIoRequest::Send и использовать получатель ввода/вывода. Создавая получатель ввода/вывода, драйвер получает доступ к интерфейсу IWDFIoTargetStateManagement, посредством которого он может запрашивать о состоянии получателя ввода/вывода и управлять им.

Кроме этого, механизм получателей ввода/вывода позволяет драйверу зарегистрировать функцию обратного вызова для завершения ввода/вывода, которую инфраструктура вызывает, когда получатель завершает запрос ввода/вывода. Драйверу не требуется выполнять опрос для завершения или обрабатывать завершение в отдельном потоке. Доступ к такому уровню контроля имеет важность для драйверов, которые должны удовлетворять требованиям Windows для своевременного завершения ввода/вывода. Даже для выполнения файлового ввода/вывода драйвер всегда должен применять получатель ввода/вывода FileHandle,

даже если он не использует никаких других получателей ввода/вывода. Применение получателя ввода/вывода FileHandle означает, что поток драйвера не блокируется при ожидании порта завершения ввода/вывода.

## Для драйверов UMDF, исполняющихся на Windows XP

Механизм получателей ввода/вывода предоставляет значительное преимущество над Windows API для драйверов, исполняющихся под Windows XP, а именно возможность отменять отдельные запросы ввода/вывода. Windows XP поддерживает только функцию CancelIo, которая отменяет все незаконченные запросы ввода/вывода для целевого файла. Драйвер UMDF, использующий получателей ввода/вывода, может отменить отдельный запрос ввода/вывода с помощью метода IWDFIoRequest::CancelSentRequest. В Windows Vista и более поздних версиях для отмены запросов применяется функция CancelIoEx, которая предоставляет возможность отмены запросов по отдельности. Соответственно, для отмены отдельного запроса ввода/вывода драйвер, исполняющийся под Windows Vista или более поздней версией, может просто вызвать функцию CancelIoEx.

Если драйвер UMDF вызывает Windows API, необходимо придерживаться следующих рекомендаций.

- ◆ Не создавайте пользовательского интерфейса. Не применяйте никаких функций, полагающихся на пользовательский интерфейс или функции, которые создают, управляют или общаются с окнами.  
Драйвер не должен полагать присутствие монитора или конечного пользователя, т. к. драйверы UMDF исполняются в сессии без присутствия пользователя.
- ◆ Всегда применяйте интерфейсы инфраструктуры для отправления запросов ввода/вывода внутри стека устройств.

Не применяйте функции ввода/вывода из Windows API для отправки синхронных запросов ввода/вывода другим драйверам в стеке устройств UMDF, т. к. это может вызвать взаимоблокировки.

- ◆ При любой возможности используйте асинхронный ввод/вывод с тем, чтобы избежать блокирования рабочего потока.
- ◆ Принимайте во внимание влияние, которое может оказывать на безопасность вызов драйвером любой функции вне пределов инфраструктуры.

## Применение вспомогательных процедур режима ядра в драйверах KMDF

Иногда драйверам KMDF требуется использовать вспомогательные процедуры режима ядра, также называемые интерфейсами WDM, не являющимися частью инфраструктуры. Наиболее широко применяемыми из таких функций являются следующие:

- ◆ функции семейств ExInterlockedXxx и InterlockedXxx, которые выполняют синхронизированные, атомарные операции, такие как, например, увеличение совместно используемой переменной;
- ◆ процедуры библиотек RTL (Run-time library, библиотека времени исполнения), преобразовывающие данные и управляющие списками.

В общем, эти широко применяемые функции не обращаются и не манипулируют объектами режима ядра, такими как DEVICE\_OBJECT или IRP. Драйверы KMDF могут использовать такие

функции режима ядра без ограничений, если вызывающая функцию драйверная процедура исполняется на нужном уровне IRQL.

Но определенные другие вспомогательные процедуры драйверов режима ядра используют объекты WDM. Драйверы KMDF могут вызывать такие процедуры и использовать объекты WDM, но должны соответствовать всем требованиям WDM для такого доступа.

### Полезная информация

На момент версии WDK Build 6000, единственным ограничением на использование объектов WDM является неприменение последних двух элементов поля `Tail.Overlay.DriverContext` в структуре IRP, т. к. эти элементы предназначены для использования инфраструктурой.

KMDF предоставляет множество методов, имена которых начинаются с WDM, которые драйвер может вызывать, чтобы получить доступ к объекту WDM. Например, драйвер может получить доступ к пакету IRP, лежащему в основе объекта типа `WDFREQUEST`, к объекту устройства WDM, лежащему в основе объекта типа `WDFDEVICE`, и т. д. В табл. 14.1 приведен список методов, предоставляющих доступ к объектам WDM.

**Таблица 14.1. Методы для доступа к объектам WDM**

Метод	Описание
<code>WdfDeviceWdmGetAttachedDevice</code>	Возвращает указатель на следующий нижележащий <code>DEVICE_OBJECT</code> WDM в стеке устройств
<code>WdfDeviceWdmGetDeviceObject</code>	Возвращает указатель на <code>DEVICE_OBJECT</code> WDM, ассоциированный с указанным инфраструктурным объектом устройства
<code>WdfDeviceWdmGetPhysicalDevice</code>	Возвращает указатель на <code>DEVICE_OBJECT</code> WDM, представляющий объект PDO для устройства
<code>WdfDpcWdmGetDpc</code>	Возвращает указатель на структуру KDPC, ассоциированную с указанным инфраструктурным объектом DPC
<code>WdfFdoInitWdmGetPhysicalDevice</code>	Возвращает указатель на <code>DEVICE_OBJECT</code> WDM, представляющий объект PDO для устройства
<code>WdfFileObjectWdmGetFileObject</code>	Возвращает указатель на <code>FILE_OBJECT</code> WDM, ассоциированный с указанным инфраструктурным файловым объектом
<code>WdfInterruptWdmGetInterrupt</code>	Возвращает указатель на структуру KINTERRUPT, ассоциированную с указанным инфраструктурным объектом прерывания
<code>WdfIoTargetWdmGetTargetDeviceObject</code>	Возвращает указатель на <code>DEVICE_OBJECT</code> WDM, ассоциированный с локальным или удаленным получателем ввода/вывода
<code>WdfIoTargetWdmGetTargetFileHandle</code>	Возвращает дескриптор файла, ассоциированного с удаленным получателем ввода/вывода
<code>WdfIoTargetWdmGetTargetFileObject</code>	Возвращает указатель на <code>FILE_OBJECT</code> WDM, ассоциированный с удаленным получателем ввода/вывода
<code>WdfIoTargetWdmGetTargetPhysicalDevice</code>	Возвращает указатель на <code>DEVICE_OBJECT</code> WDM, представляющий объект PDO для удаленного получателя ввода/вывода

Таблица 14.1 (окончание)

Метод	Описание
WdfRegistryWdmGetHandle	Возвращает дескриптор WDM для раздела реестра, представляемый инфраструктурным объектом раздела реестра
WdfRequestCreateFromIrp	Создает объект WDFREQUEST для представления пакета IRP WDM
WdfRequestRetrieveInputWdmMdl	Возвращает указатель на список MDL, представляющий буфер ввода запроса ввода/вывода
WdfRequestRetrieveOutputWdmMdl	Возвращает указатель на список MDL, представляющий буфер вывода запроса ввода/вывода
WdfRequestWdmGetIrp	Возвращает указатель на пакет IRP WDM, ассоциированный с указанным инфраструктурным объектом запроса
WdfUsbTargetPipeWdmGetPipeHandle	Возвращает дескриптор типа USBD_PIPE_HANDLE, ассоциированный с указанным инфраструктурным объектом канала
WdfWdmDeviceGetWdfDeviceHandle	Возвращает дескриптор инфраструктурного объекта устройства, ассоциированного с указанным DEVICE_OBJECT WDM
WdfWdmDriverGetWdfDriverHandle	Возвращает дескриптор инфраструктурного объекта драйвера, ассоциированного с указанным DRIVER_OBJECT WDM

Драйверу KMDF может потребоваться доступ к нижележащим в стеке объектам WDM по многим причинам. Наиболее часто драйверу может потребоваться указатель на подлегающий объект устройства WDM, чтобы открыть интерфейс для драйвера в другом стеке устройств или чтобы вызвать функцию режима ядра, отсутствующую в KMDF, для выполнения какой-либо задачи, связанной с устройством, как, например, регистрация определенных типов извещений от устройства.

Кроме этого, информация WDM часто может быть полезной при выполнении отладки и тестирования драйвера. Многие из существующих инструментов для тестирования, например, Driver Verifier, предоставляют информацию, используя базовые объекты ядра, а не объекты WDF. Сохранение значений объектов DRIVER\_OBJECT и DEVICE\_OBJECT в журнале трассировки может помочь сэкономить время при отладке.

В листинге 14.2 приводится фрагмент кода из образца драйвера Toastmon, являющегося частью комплекта драйверов KMDF. В нем показывается, каким образом драйвер KMDF вызывает функцию менеджера ввода/вывода и передает ей указатель на объект DEVICE\_OBJECT WDM.

#### Листинг 14.2. Вызов функции менеджера ввода/вывода в драйвере KMDF

```
status = IoRegisterPlugPlayNotification(
    EventCategoryDeviceInterfaceChange,
    PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES,
    (PVOID)&GUID_DEVINTERFACE_TOASTER,
    WdfDriverWdmGetDriverObject(WdfDeviceGetDriver(device)),
```

```
(PDRIVER_NOTIFICATION_CALLBACK_ROUTINE)
    ToastMon_PnpNotifyInterfaceChange,
(PVOID)deviceExtension,
&deviceExtension->NotificationHandle);
```

Драйвер вызывает функцию `IoRegisterPlugPlayNotification` менеджера ввода/вывода Windows, чтобы запросить извещение, когда в его стеке устройств происходят определенные события Plug and Play. Образец драйвера `Toastmon` использует эту функцию, чтобы зарегистрироваться для извещений об изменениях интерфейса устройства, чтобы система извещала его при каждом подключении или отключении устройства. В этой функции первые три параметра описывают событие, о котором драйвер требует извещения. Четвертый параметр — это указатель на объект `DRIVER_OBJECT` WDM вызывающего клиента. Драйвер KMDF получает этот указатель в двухэтапном процессе:

1. Вызывает метод `wdfDeviceGetDriver`, передавая ему дескриптор объекта `WDFDEVICE`, чтобы получить дескриптор объекта `WDFDRIVER`.
2. Вызывает метод `wdfDriverWdmGetDriverObject`, передавая ему только что полученный дескриптор объекта `WDFDRIVER`, чтобы получить указатель на объект `DRIVER_OBJECT` WDM.

Пятый параметр — указатель на функцию обратного вызова, которую система активирует, когда происходит указанное событие Plug and Play. Шестой параметр — указатель на информацию контекста, которая передается функции обратного вызова. Наконец, последний параметр — это идентификатор регистрации, который драйвер использует позже для отмены регистрации.

## Обработка запросов, не поддерживаемых инфраструктурами

Ни UMDF, ни KMDF не поддерживают все возможные типы запросов ввода/вывода. Например, инфраструктуры не поддерживают запросы файловой системы, такие как `IRP_MJ_FILE_SYSTEM_CONTROL`. Вместо этого, инфраструктуры поддерживают типы запросов ввода/вывода, наиболее часто получаемые драйверами.

Полный список поддерживаемых типов запросов приводится в главе 8.

## Обработка по умолчанию неподдерживаемых запросов

Обработка по умолчанию неподдерживаемых запросов выполняется по-разному в UMDF и KMDF.

Для драйверов UMDF, когда отражатель получает пакет IRP неподдерживаемого типа, он передает его следующему нижележащему драйверу в стеке устройств режима ядра. Драйверы UMDF не имеют доступа к запросам, не поддерживаемым инфраструктурами.

Когда инфраструктура KMDF получает запрос неподдерживаемого типа, действие, которое она предпринимает, зависит от типа объекта устройства, который был получателем запроса. Для объекта FDO или PDO инфраструктура завершает пакет IRP, возвращая статус `STATUS_INVALID_DEVICE_REQUEST`. Для объекта FiDO инфраструктура передает пакет IRP следующему нижележащему драйверу.

Например, драйвер KMDF может переопределить это поведение по умолчанию, регистрируя функцию обратного вызова, которая вызывается при прибытии неподдерживаемого пакета IRP, которая и обрабатывает этот пакет.

## Обработка неподдерживаемых запросов в драйверах KMDF

KMDF определяет функцию обратного вызова для предварительной обработки пакета IRP WDM, с помощью которой драйвер может получать любой неподдерживаемый пакет IRP WDM. Для обработки таких пакетов IRP:

- ◆ в функции обратного вызова *EvtDriverDeviceAdd* вызывается метод *WdfDeviceInitAssignWdmIrpPreprocessCallback* для каждого типа пакета IRP WDM, поддерживаемого драйвером.

При вызове метода указывается имя функции обратного вызова *EvtDeviceWdmIrpPreprocess*, код основной функции пакета IRP и коды любых дополнительных функций, обрабатываемых функцией обратного вызова. Драйвер должен зарегистрировать свои функции обратного вызова до создания объекта устройства;

- ◆ в функции обратного вызова *EvtDeviceWdmIrpPreprocess* выполняются все операции, требуемые драйвером для обработки пакета IRP, используя функции WDM требуемым образом.

В каждом вызове метода *WdfDeviceInitAssignWdmIrpPreprocessCallback* регистрируется функция обратного вызова для предварительной обработки для одного основного типа пакета IRP. Драйвер может вызывать этот метод столько раз, сколько необходимо, чтобы зарегистрировать функции обратного вызова для нескольких типов пакетов IRP. Одна функция обратного вызова может выполнять предварительную обработку нескольких типов пакетов IRP.

Инфраструктура не синхронизирует вызов функции *EvtDeviceWdmIrpPreprocess* с состояниями Plug and Play и энергопотребления объекта устройства, поэтому драйверу не гарантируется наличие аппаратного обеспечения. Кроме этого, инфраструктура не засчитывает пакет IRP в своем подсчете запросов ввода/вывода для объекта устройства. Поэтому при удалении устройства или уничтожении объекта устройства инфраструктура не ожидает завершения пакета IRP. Эту синхронизацию драйвер должен обеспечить собственными силами.

По окончанию обработки пакета IRP функцией *EvtDeviceWdmIrpPreprocess* функция *EvtDeviceWdmIrpPreprocess* выполняет одно из следующих действий.

- ◆ Вызывает процедуру *IoCompleteRequest* для завершения пакета IRP.
- ◆ Организует следующую ячейку стека ввода/вывода, вызывая макрос *IoSkipCurrentIrpStackLocation* или процедуру *IoCopyCurrentIrpStackLocationToNext*, и вызывает процедуру *IoCallDriver*, чтобы передать пакет IRP следующему нижележащему драйверу, точно так же, как это бы сделал драйвер WDM.
- ◆ Вызывает метод *WdfDeviceWdmDispatchPreprocessedIrp*, чтобы передать пакет IRP инфраструктуре для обработки обычным образом.

Для объекта FiDO инфраструктура передает пакет IRP следующему нижележащему драйверу; для объекта любого другого устройства инфраструктура завершает пакет IRP неудачей, как описано в разд. "Обработка по умолчанию неподдерживаемых запросов" ранее в этой главе.

Если драйвер или инфраструктура передает пакет IRP вниз по стеку устройств, но драйверу требуется доступ к пакету IRP после его завершения нижележащими драйверами (т. е. драйвер обрабатывает пакет IRP после его завершения), то перед тем, как передать пакет IRP следующему нижележащему драйверу, функция *EvtDeviceWdmIrpPreprocess* должна вызвать

для пакета процедуру `IoSetCompletionRoutine` или процедуре `IoSetCompletionRoutineEx`. Процедура `IoCompletion` завершает обработку операций ввода/вывода. Функция обратного вызова `EvtDeviceWdmIrpPreprocess`, процедуры `IoSetCompletionRoutine` и `IoSetCompletionRoutineEx` и процедура `IoCompletion` должны следовать обычным правилам WDM для обработки пакетов IRP.

### Пример: организация функции обратного вызова предварительной обработки `EvtDriverDeviceAdd`

В образце драйвера Serial показано, как реализовывать и применять функцию обратного вызова `EvtDeviceWdmPreprocessIrp` для обработки и завершения запросов `IRP_MJ_FLUSH_BUFFERS`, `IRP_MJ_QUERY_INFORMATION` и `IRP_MJ_SET_INFORMATION`. В листинге 14.3 приведен фрагмент кода из файла `KmdfSerial\Pnp.c`, в котором показано, как функция обратного вызова `EvtDriverDeviceAdd` драйвера регистрирует функцию обратного вызова `EvtDeviceWdmIrpPreprocess` для запроса `IRP_MJ_FLUSH_BUFFERS`.

#### Листинг 14.3. Регистрация функции обратного вызова `EvtDeviceWdmPreprocessIrp`

```
status = WdfDeviceInitAssignWdmIrpPreprocessCallback(
    DeviceInit,
    SerialFlush,
    IRP_MJ_FLUSH_BUFFERS,
    NULL, // Указатель на таблицу дополнительных функций.
    0); // Количество элементов в таблице.
```

Образец драйвера регистрирует функцию обратного вызова в процессе инициализации устройства, прежде чем он создает объект устройства. Методу `WdfDeviceInitAssignWdmIrpPreprocessCallback` передаются следующие пять параметров:

- ◆ указатель на `PWDFDEVICEINIT`, который был передан функции `EvtDriverDeviceAdd(DeviceInit)`;
- ◆ указатель на функцию обратного вызова `EvtDeviceWdmIrpPreprocess`, которую нужно зарегистрировать (`SerialFlush`);
- ◆ код основной функции пакета IRP, для которого нужно вызвать функцию обратного вызова (`IRP_MJ_FLUSH_BUFFERS`);
- ◆ указатель на массив кодов дополнительных функций, для которых требуется активизировать функцию обратного вызова. В данном случае это `NULL`, т. к. функция `IRP_MJ_FLUSH_BUFFERS` не имеет кодов дополнительных функций;
- ◆ значение `ULONG`, указывающее количество элементов в массиве (0).

Всякий раз, когда для объекта устройства прибывает запрос `IRP_MJ_FLUSH_BUFFERS`, инфраструктура активизирует функцию `SerialFlush` обратного вызова драйвера.

### Пример: обработка в функции обратного вызова `EvtDeviceWdmPreprocessIrp`

В листинге 14.4 приводится фрагмент кода из файла `KmdfSerial\Flush.c`, представляющий функцию обратного вызова `EvtDeviceWdmPreprocessIrp`, зарегистрированную в предыдущем примере. При вызове этой функции передается дескриптор объекта `WDFDEVICE` и указатель на пакет IRP WDM.

**Листинг 14.4. Обработка пакета IRP WDM в функции обратного вызова EvtDeviceWdmPreprocessIrp**

```
NTSTATUS SerialFlush(IN WDFDEVICE Device,
                     IN PIRP Irp)
{
    PSERIAL_DEVICE_EXTENSION extension;
    extension = SerialGetDeviceExtension(Device);
    WdfIoQueueStopSynchronously(extension->WriteQueue);
    // Сброс выполнен, перезапускаем очередь.
    WdfIoQueueStart(extension->WriteQueue);
    Irp->IoStatus.Information = 0L;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

В представленном листинге показана обработка запроса `IRP_MJ_FLUSH_BUFFERS`. Этот запрос не имеет параметров, поэтому драйвер сразу же начинает его обработку. Для данного стека устройств запрос на сброс буферов указывает запрос на сброс очереди запросов на запись. Чтобы выполнить требуемое действие, драйвер вызывает метод `WdfIoQueueStopSynchronously`, который возвращает управление, когда все запросы в очереди завершены или отменены. После этого драйвер перезапускает очередь и завершает пакет IRP.

Для завершения пакета IRP драйвер записывает значение `NTSTATUS` в поле `IoStatus.Status`, а число переданных байтов — в поле `IoStatus.Information` пакета IRP. После этого он вызывает функцию `IoCompleteRequest` менеджера ввода/вывода, которая и завершает пакет IRP. Драйвер не может использовать метод `WdfRequestComplete`, т. к. инфраструктура не создала объект `WDFREQUEST` для представления запроса. По этой же причине, если драйвер не завершил запрос, но переслал его вниз по стеку, он не может использовать получатель ввода/вывода.

Если драйвер обрабатывает пакеты IRP вне пределов инфраструктуры, как показано в данном примере, он должен следовать обычным правилам WDM и использовать функции WDM для манипулирования пакетом IRP и любыми другими структурами данных WDM.

Для углубленного рассмотрения принципов и правил обработки пакетов IRP в драйверах WDF см. раздел **Kernel-Mode Driver Architecture Design Guide** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79288>.

# ГЛАВА 15

## Планирование, контекст потока и уровни IRQL

На работу драйверов режима ядра большое влияние оказывают такие факторы, как контекст потока и текущий уровень прерывания (IRQL) для каждого процессора. Исчерпывающее описание планирования потоков и временных квантов приводится в книге "Microsoft Windows Internals"<sup>1</sup>. В данной главе предоставляются основные понятия планирования потоков, контекста потоков и уровней IRQL. Также в ней объясняется, каким образом драйвер может использовать рабочий элемент, чтобы запустить функцию, исполняющуюся на уровне IRQL = PASSIVE\_LEVEL, из кода, исполняющегося на уровне IRQL = DISPATCH\_LEVEL.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Usbsamp	%wdk%\src\kmdf\usbsamp
Toastmon	%wdk%\src\kmdf\Toaster\Toastmon
<b>Документация WDK</b>	
Do Waiting Threads Receive Alerts and APCs? <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80071">http://go.microsoft.com/fwlink/?LinkId=80071</a>
Specifying Priority Boosts When Completing I/O Requests <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=81582">http://go.microsoft.com/fwlink/?LinkId=81582</a>
Locking Pageable Code or Data <sup>4</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=82719">http://go.microsoft.com/fwlink/?LinkId=82719</a>
<b>Прочее</b>	
Раздел <b>Interrupt Architecture Enhancements in Windows</b> <sup>5</sup> на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=81584">http://go.microsoft.com/fwlink/?LinkId=81584</a>
Книга "Microsoft Windows Internals"	<a href="http://go.microsoft.com/fwlink/?LinkId=82721">http://go.microsoft.com/fwlink/?LinkId=82721</a>

<sup>1</sup> Внутреннее устройство Microsoft Windows 2003, Windows XP и Windows 2000. — Издательско-торговый дом "Русская Редакция"; СПб.: Питер, 2005.

<sup>2</sup> Получают ли потоки, находящиеся в состоянии ожидания, извещения и процедуры ACP? — *Пер.*

<sup>3</sup> Указание форсажа (повышения) приоритета при завершении запросов ввода/вывода. — *Пер.*

<sup>4</sup> Блокировка страничного кода или данных. — *Пер.*

<sup>5</sup> Улучшения в архитектуре прерываний Windows. — *Пер.*

## Потоки

Поток является базовой единицей исполнения в Windows. Но для исполнения драйверов KMDF Windows не создает потоков, и большинство драйверов KMDF также не создают потоков. Вместо этого, драйвер, который представляет собой группу функций, вызывается в уже существующем потоке, который "берется взаимо" у приложения или системного компонента. Но контекст этого существующего потока накладывает ограничение на доступ драйвера к данным. Поэтому при создании драйверов, исполняющих определенные типы операций ввода/вывода, важно понимать контекст потока, в котором исполняются некоторые функции обратного вызова драйверов.

## Планирование потоков

В Windows для исполнения планируются отдельные потоки, а не целые процессы. Каждому потоку назначается приоритет планирования, или приоритет потока, в виде значений от 1 до 31 включительно. Более высокий приоритет указывается более высоким числом.

Каждому потоку выделяется временной квант, который определяет максимальное количество времени центрального процессора, в течение которого поток может исполняться, прежде чем ядро начнет рассматривать для исполнения другие потоки с таким же приоритетом. Длительность кванта времени зависит от установленной версии Windows, типа процессора и заданных системных настроек производительности.

Запланированный поток исполняется, пока не произойдет одно из следующих:

- ◆ выделенный ему квант времени истек;
- ◆ поток входит в состояние ожидания;
- ◆ становится готовым к исполнению поток с более высоким приоритетом.

Потоки режима ядра не имеют приоритета над потоками пользовательского режима, и поток режима ядра может быть вытеснен потоком пользовательского режима, имеющим более высокий приоритет.

Приоритеты потока, имеющие значения в диапазоне от 1 до 15, называются *динамическими*. Приоритеты потока, имеющие значения в диапазоне от 16 до 31, называются приоритетами *реального времени*. Поток с нулевым приоритетом зарезервирован для обнуления неиспользуемых страниц памяти, если менеджеру памяти требуется свободная память.

Каждый поток имеет базовый приоритет и текущий приоритет.

- ◆ базовый приоритет обычно наследуется от базового приоритета процесса потока;
- ◆ текущий приоритет — это приоритет потока в данный момент времени.

Базовым приоритетом кода драйвера режима ядра, исполняющегося в контексте пользовательского потока, является приоритет пользовательского процесса, первоначально запросившего операцию ввода/вывода. Базовым приоритетом кода драйвера режима ядра, исполняющегося в контексте системного рабочего потока, например рабочего элемента, является базовый приоритет системных рабочих потоков, обслуживающих его очередь.

Чтобы улучшить пропускную способность системы, Windows иногда корректирует приоритеты потоков. Так, Windows может временно повысить (т. е. форсировать) или понизить текущий приоритет потока с базовым приоритетом в динамическом диапазоне, в результате чего текущий приоритет будет отличаться от базового приоритета. Базовый и текущий при-

оритет потоков с приоритетом в диапазоне реального времени всегда одинаковый, т. е. приоритет потоков реального времени никогда не форсируется. Более того, динамический приоритет потока не может быть повышен до приоритета реального времени. Таким образом, потоки с базовым приоритетом в диапазоне приоритетов реального времени всегда имеют более высокий приоритет, чем потоки с базовым приоритетом в динамическом диапазоне.

Windows может повысить приоритет потока, когда поток завершает запрос ввода/вывода, когда он завершает ожидание события или семафора или когда он не исполнялся в течение некоторого времени, несмотря на его готовность к исполнению (ситуация, называемая "процессорное голодание" (CPU starvation)). Приоритет потоков, принимающих участие в обеспечении графического интерфейса (GUI) и в пользовательских приоритетных процессах, также может повышаться в определенных ситуациях. Величина повышения приоритета зависит от вызвавшей ее причины, а для операций ввода/вывода — от типа вовлеченного устройства.

Драйверы могут оказывать влияние на повышение приоритета своего кода, указывая следующее:

- ◆ повышение приоритета в вызове метода `WdfRequestCompleteWithPriorityBoost`;
- ◆ инкремент приоритета в вызове процедуры `KeSetEvent`, `KePulseEvent` или `KeReleaseSemaphore`.

Соответствующие повышения приоритета для каждого типа устройства, события и семафора указываются константами, определенными в заголовочном файле `Wdm.h`. По умолчанию инфраструктура применяет форсаж приоритета по умолчанию на основе типа устройства, даже если драйвер завершает запрос, вызывая метод `WdfRequestComplete`.

Дополнительную информацию см. в разделе **Specifying Priority Boosts When Completing I/O Requests** (Задание повышений приоритета при завершении запросов ввода/вывода) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81582>.

### Примечание

Понятие уровня приоритета планирования потока имеет совершенно иной смысл, чем понятие уровня IRQL процессора при исполнении этого потока.

## Определение контекста потока

Разработчики программного обеспечения режима ядра применяют термин "контекст потока" для обозначения двух слегка разных понятий. В самом узком значении этого термина, *контекст потока* — это значение структуры `CONTEXT` потока. Структура `CONTEXT` содержит значения аппаратных регистров, стеков и локальных областей памяти потока. Специфическое содержимое и организация этой структуры зависят от аппаратной платформы. Когда Windows планирует пользовательский поток, она загружает информацию из структуры `CONTEXT` потока в адресное пространство пользовательского режима.

Но применительно к разработке драйверов термин "контекст потока" имеет более широкое значение. Для драйвера контекст потока означает не только значения, хранящиеся в структуре `CONTEXT`, но также определяемую ими операционную среду, в особенности права безопасности вызывающего приложения. Например, функция драйвера может быть вызванной в контексте потока пользовательского режима, но в свою очередь может вызвать функцию семейства `zwXXX` для выполнения операции в контексте ядра операционной системы. В этой главе термин "контекст потока" применяется в более широком значении.

Когда функция драйвера исполняет операцию ввода/вывода, контекст потока может содержать адресное пространство пользовательского режима и права безопасности процесса, запросившего ввод/вывод. Но если вызывающий процесс исполнял операцию для другого пользователя или приложения, контекст потока может содержать адресное пространство пользовательского режима и права безопасности иного процесса. Другими словами, адресное пространство пользовательского режима может содержать информацию, относящуюся к процессу, запросившему ввод/вывод, или же оно может содержать информацию, принадлежащую совсем другому процессу.

## Контекст потока драйверных функций KMDF

Контекст потока, в котором вызываются драйверные функции KMDF, зависит от типа устройства, позиции драйвера в стеке и от другой деятельности, происходящей в данный момент в системе. Функции обратного вызова KMDF исполняются в одном из следующих трех контекстов потока:

- ◆ системного потока;
- ◆ вызывающего потока;
- ◆ произвольного потока.

Некоторые драйверные функции исполняются в контексте системного потока, который обладает адресным пространством системного процесса и правами безопасности самой операционной системы. Все процедуры `DriverEntry` и функции обратного вызова `EvtDriverDeviceAdd` исполняются в контексте системного потока, как и обратные вызовы рабочих элементов, поставленные в очередь методом `WdfWorkItemEnqueue`. В контексте системного потока не прибывают никакие запросы пользовательского режима.

Рабочие функции ввода/вывода верхнего драйвера стека устройств — часто это драйвер файловой системы WDM или драйвер фильтра файловой системы — получают запросы ввода/вывода в контексте потока, инициировавшего запрос. Эти функции могут обращаться к данным в адресном пространстве пользовательского режима запрашивающего процесса, но они должны проверить достоверность указателей и принять меры предосторожности, чтобы уберечь себя от ошибок пользовательского режима.

Большинство других драйверных функций вызываются в контексте произвольного потока. Хотя верхний драйвер в стеке получает запросы ввода/вывода в контексте запрашивающего потока, обычно этот драйвер пересыпает эти запросы нижележащим драйверам, которые исполняются в иных потоках.

Например, когда приложение пользовательского режима запрашивает операцию синхронного ввода/вывода, менеджер ввода/вывода вызывает рабочую функцию ввода/вывода верхнего драйвера в стеке устройств режима ядра в контексте потока, запросившего операцию. Эта рабочая функция ставит запрос ввода/вывода в очередь для обработки драйверами низших уровней. После этого запрашивающий поток переходит в состояние ожидания, в котором он пребывает до завершения запроса ввода/вывода. Запрос убирается из очереди другим процессом и обрабатывается нижележащими драйверами, которые исполняются в контексте потока, исполняющегося при их вызове.

Соответственно, нельзя делать никаких предположений относительно содержимого адресного пространства пользовательского режима или относительно контекста потока, в котором исполняется большинство функций обратного вызова драйверов KMDF. Но в некото-

рых случаях необходимо организовать исполнение драйверной функции в контексте вызывающего ее клиента. Если драйвер KMDF не исполняет ни буферизированный, ни прямой ввод/вывод (обычно называемый neither I/O или METHOD\_NEITHER I/O), он получает указатель буфера пользовательского режима в запросе IOCTL. Драйвер может получить доступ к этому указателю только в контексте потока, издавшего запрос. Поэтому, чтобы получить доступ к этому указателю, драйвер предоставляет функцию обратного вызова *EvtIoInCallerContext*, которую инфраструктура вызывает в контексте потока, вызвавшего драйвер. Если драйвер KMDF является верхним драйвером в своем стеке устройств, при соблюдении должных мер предосторожности код в функции обратного вызова может получить доступ к указателю буфера пользовательского режима, чтобы заблокировать память этого буфера. Но если этот драйвер KMDF находится в стеке устройств под одним или несколькими драйверами фильтра, клиент, вызывающий этот драйвер, может не быть потоком пользовательского режима, который инициировал запрос, и указатель может быть недействительным.

Дополнительная информация о функции обратного вызова *EvtIoInCallerContext* и пример ее кода предоставлены в главе 8.

## Уровни IRQL

Уровень IRQL определяет приоритет аппаратного средства, на котором работает процессор в любой данный момент.

Когда процессор работает на определенном уровне IRQL, прерывания этого и более низких уровней маскируются от процессора.

Поток, исполняемый на низком уровне IRQL, можно прервать для исполнения кода с более высоким уровнем IRQL, но поток, исполняющийся на высоком уровне IRQL, нельзя прервать для исполнения кода с таким же или более низким уровнем IRQL. Например, процессор, работающий на уровне IRQL = DISPATCH\_LEVEL, может прервать только запрос с уровнем IRQL > DISPATCH\_LEVEL.

Число уровней IRQL и их конкретные значения зависят от процессора. Архитектуры x64 и Intel Itanium имеют 16 уровней IRQL, а архитектуры на основе x86 имеют 32 уровня IRQL. Разница в основном обуславливается типом контроллера, используемым в каждой архитектуре. В табл. 15.1 приведен список уровней IRQL для процессоров архитектуры x86, x64 и Intel Itanium.

**Таблица 15.1. Уровни IRQL для разных типов процессоров**

IRQL	Значение IRQL для конкретного процессора			Исполняемые операции
	x86	x64	Itanium	
PASSIVE_LEVEL	0	0	0	Пользовательские потоки и большинство операций режима ядра
APC_LEVEL	1	1	1	Вызовы асинхронных процедур и обработка ошибок страницы
DISPATCH_LEVEL	2	2	2	Планировщик потоков и процедуры DPC
CMC_LEVEL	N/A	N/A	3	Уровень исправимых машинных сбоев (только на платформах Itanium)

Таблица 15.1 (окончание)

IRQL	Значение IRQL для конкретного процессора			Исполняемые операции
	x86	x64	Itanium	
Уровни прерываний устройств (DIRQL)	3-26	3-11	4-11	Прерывания устройств
PC_LEVEL	N/A	N/A	12	Счетчик производительности (только на платформах Itanium)
PROFILE_LEVEL	27	15	15	Профильный таймер для версий Windows более ранних, чем Windows 2000
SYNCH_LEVEL	27	13	13	Синхронизация кода и потоков инструкций по процессорам
CLOCK_LEVEL	N/A	13	13	Таймер
CLOCK2_LEVEL	28	N/A	N/A	Таймер для аппаратного обеспечения x86
IPI_LEVEL	29	14	14	Межпроцессорное прерывание для обеспечения непротиворечивости кэшей
POWER_LEVEL	30	15	15	Нарушение энергоснабжения
HIGH_LEVEL	31	15	15	Машинные сбои и фатальные ошибки; профильный таймер для Windows XP и более поздних версий

Система планирует все потоки для исполнения на уровнях IRQL ниже уровня DISPATCH\_LEVEL, и сам системный планировщик потоков (также называемый *диспетчером*) исполняется на уровне DISPATCH\_LEVEL. Таким образом, поток, исполняющийся на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , в сущности, пользуется текущим процессором на эксклюзивной основе. Так как прерывания уровня DISPATCH\_LEVEL маскируются от процессора, планировщик потоков не может исполняться на этом процессоре и, соответственно, не может запланировать на исполнение никаких других потоков.

На многопроцессорной системе каждый из процессоров может работать на разном уровне IRQL. Поэтому один процессор может исполнять функцию *EvtInterruptIsr* драйвера на уровне DIRQL, в то время как другой процессор исполняет код драйвера в рабочем потоке на уровне PASSIVE\_LEVEL. Таким образом, несколько потоков могут пытаться получить доступ к разделяемым данным одновременно. Если потоки только считывают данные, то никаких блокировок не требуется. Но если хотя бы один из потоков записывает в эти данные, драйвер должен сериализовать доступ, используя блокировку, которая повышает уровень IRQL до самого высокого уровня, на котором может исполняться любой код, обращающийся к данным. В данном примере код, исполняющийся на уровне PASSIVE\_LEVEL в рабочем потоке, прежде чем обращаться к разделяемым данным, захватывает спин-блокировку.

## Уровни IRQL, специфичные для процессора и потока

Уровни IRQL можно рассматривать как специфичные для процессора или потока. Уровни выше DISPATCH\_LEVEL являются специфичными для процессора. Аппаратные и программные прерывания на этих уровнях нацелены на отдельные процессоры.

Драйверы широко используют следующие специфичные для процессора уровни IRQL:

- ◆ DISPATCH\_LEVEL;
- ◆ DIRQL;
- ◆ HIGHEST\_LEVEL.

Уровни IRQL ниже DISPATCH\_LEVEL являются специфичными для потока. Программные прерывания на этих уровнях нацелены на отдельные потоки. Драйверы используют следующие специфичные для потока уровни IRQL:

- ◆ PASSIVE\_LEVEL;
- ◆ APC\_LEVEL.

При вытеснении потока системный планировщик потоков принимает во внимание только приоритет потока, а не уровень IRQL. Если поток исполняется в блоках уровня IRQL = APC\_LEVEL, то планировщик может выбрать для процессора новый поток, который раньше исполнялся на уровне PASSIVE\_LEVEL.

Хотя определено только два значения уровня IRQL, специфичных для потока, в действительности система реализует три уровня. Кроме уровней PASSIVE\_LEVEL и APC\_LEVEL, система реализует промежуточный уровень между этими двумя уровнями. О коде, исполняющемся на этом уровне, говорят, что он находится в критической области. Чтобы поднять свой уровень IRQL до этого значения, код, исполняющийся на уровне PASSIVE\_LEVEL, вызывает процедуру KeEnterCriticalSection, а чтобы возвратиться обратно на уровень PASSIVE\_LEVEL — процедуру KeLeaveCriticalSection.

В следующих разделах приводится дополнительная информация об операционной среде для кода драйвера на каждом из этих уровней.

## Уровень IRQL PASSIVE\_LEVEL

Когда процессор работает на уровне PASSIVE\_LEVEL, Windows принимает решения о том, какие потоки выполнять, на основе приоритетов планирования текущих потоков. Нормальным рабочим состоянием процессора является уровень PASSIVE\_LEVEL. Любой поток, исполняющийся на уровне PASSIVE\_LEVEL, считается вытесняемым, т. к. он может быть заменен потоком, имеющим более высокий приоритет планирования. Также поток, исполняющийся на уровне PASSIVE\_LEVEL, считается прерываемым, т. к. он может быть прерван запросом с более высоким уровнем IRQL.

Иногда код драйвера, исполняющийся на уровне PASSIVE\_LEVEL, должен вызывать функции системного обслуживания или выполнять какие-либо другие операции, требующие исполнения на более высоком уровне IRQL, обычно на уровне DISPATCH\_LEVEL. Прежде чем вызвать такую функцию или выполнить операцию, драйвер должен повысить свой уровень IRQL до требуемого значения, и возвратить его обратно сразу же по завершению операции. Для повышения уровня IRQL драйвер вызывает процедуру KeRaiseIrql, а для понижения — KeLowerIrql.

Код, исполняющийся на уровне PASSIVE\_LEVEL, считается выполняющим работу для текущего потока. Приложение, создавшее поток, может приостановить этот поток, пока поток исполняет код режима ядра на уровне PASSIVE\_LEVEL. Поэтому для кода драйвера, который захватывает блокировку на уровне PASSIVE\_LEVEL, необходимо убедиться в том, что поток, в котором он исполняется, не может быть приостановлен, пока он удерживает блокировку; приостановка потока сделала бы невозможным доступ к устройству драйвера. Этую

проблему можно решить, используя блокировку, которая повышает уровень IRQL, или переходом в критическую область, когда драйвер пытается захватить блокировку на уровне PASSIVE\_LEVEL.

В главе 10 описывается, каким образом драйвер KMDF может принудить определенные функции обратного вызова для объектов очереди и файла исполняться на уровне IRQL = PASSIVE\_LEVEL.

### **Уровень IRQL PASSIVE\_LEVEL в критической области**

Код, исполняющийся в критической области на уровне PASSIVE\_LEVEL, по существу исполняется на промежуточном уровне между уровнями PASSIVE\_LEVEL и APC\_LEVEL. В этой ситуации процедура определения текущего уровня `KeGetCurrentIrql` возвращает значение PASSIVE\_LEVEL. Код драйвера может определить, работает ли он в критической области, вызывая функцию `KeAreApcSDisabled`, которая доступна в Windows XP и более поздних выпусках.

Процедуры APC представляют собой программные прерывания, нацеленные на определенный поток. Система применяет процедуры APC для выполнения работы в контексте определенного потока, например, для предоставления статуса операции ввода/вывода запросившему приложению. Каким образом целевой поток реагирует на процедуры APC, зависит от состояния потока и типа процедуры.

Код драйвера, исполняющийся выше уровня PASSIVE\_LEVEL — или в критической области на уровне PASSIVE\_LEVEL или на уровне  $\text{IRQL} \geq \text{APC\_LEVEL}$  — нельзя приостановить. Если драйвер устанавливает ограничение на исполнение на уровне PASSIVE\_LEVEL для функций обратного вызова объекта файла или устройства, инфраструктура синхронизирует исполнение этих функций посредством перехода в критическую область. Критическая область предотвращает возможную ситуацию DoS, которая может возникнуть в результате приостановки потока.

Почти все операции, которые драйвер может выполнять на уровне PASSIVE\_LEVEL, можно также выполнять в критической области. Два значительных исключения составляют генерация фатальных ошибок и открытие файлов на устройствах хранения.

### **Уровень IRQL APC\_LEVEL**

Уровень IRQL является уровнем специфичным для потока, который наиболее часто ассоциируется с вводом/выводом в файл подкачки. Приложения не могут приостановить код, исполняющийся на уровне APC\_LEVEL. Система реализует быстрые мьютексы — тип механизма синхронизации — на уровне APC\_LEVEL. Функция `KeAcquireFastMutex` повышает уровень IRQL до APC\_LEVEL, а функция `KeReleaseFastMutex` возвращает его в первоначальное значение.

Единственная разница между потоком, исполняющимся на уровне PASSIVE\_LEVEL с запрещением на процедуры APC, и потоком, исполняющимся на уровне APC\_LEVEL, заключается в том, что при исполнении на уровне APC\_LEVEL поток не может быть прерван специальной процедурой APC режима ядра, доставляемой системой по завершению запроса ввода/вывода.

### **Уровень IRQL DISPATCH\_LEVEL**

Уровень DISPATCH\_LEVEL — это самый высший уровень программных прерываний и первый из уровней, специфичных для процессора. Диспетчер Windows исполняется на

уровне DISPATCH\_LEVEL. Некоторые другие вспомогательные функции режима ядра, некоторые драйверные функции и все процедуры DPC также исполняются на уровне IRQL = DISPATCH\_LEVEL. Когда процессор работает на этом уровне, один поток не может быть вытеснен другим; только аппаратное прерывание может прервать поток, исполняющийся на этом уровне. Чтобы максимизировать пропускную способность системы, код драйвера, исполняющийся на уровне DISPATCH\_LEVEL, должен исполнять минимально необходимый объем работы.

Так как код, исполняющийся на уровне DISPATCH\_LEVEL, нельзя вытеснить, то драйвер может выполнять лишь ограниченные операции на этом уровне. Любой код, который должен ожидать объект, устанавливаемый или передаваемый другим потоком асинхронным образом — например событие, семафор, мьютекс или таймер — не может исполняться на уровне DISPATCH\_LEVEL, т. к. ожидающему потоку нельзя блокироваться в ожидании на другой поток выполнить операцию. Ожидание такого объекта в течение ненулевого периода на уровне DISPATCH\_LEVEL вызывает зависание и, в конечном счете, сбой системы.

По существу, процедуры DPC являются программными прерываниями, направленными на процессоры. Процедуры DPC, включая функции *EvtInterruptDpc* и *EvtDpcFunc*, всегда вызываются на уровне DISPATCH\_LEVEL в контексте произвольного потока. Драйверы обычно используют процедуры DPC для следующих целей.

- ◆ Выполнять дополнительную обработку после прерывания устройства.

Для этого применяются такие процедуры DPC, как *EvtInterruptDpc* или *EvtDpcFunc*, которые ставятся в очередь функцией обратного вызова *EvtInterruptIsr* драйвера.

- ◆ Обрабатывать тайм-ауты устройства.

Для этого применяется такая процедура DPC, как функция обратного вызова *EvtTimerFunc*, которую инфраструктура ставит в очередь по истечении таймера, запущенного методом *WdfTimerStart*.

Ядро содержит очередь процедур DPC для каждого процессора и исполняет процедуры DPC из этой очереди непосредственно перед понижением уровня IRQL процессора ниже уровня DISPATCH\_LEVEL. Процедура DPC назначается в очередь того самого процессора, но которой исполняется код, поставивший ее в очередь.

Если устройство выдает прерывание во время исполнения функции обратного вызова *EvtInterruptDpc* или *EvtDpcFunc*, его функция обратного вызова *EvtInterruptIsr* прерывает процедуру DPC и ставит объект DPC в очередь обычным способом. На однопроцессорной системе объект DPC ставится в конец одной очереди процедур DPC, из которой он исполняется в последовательности со всеми другими процедурами DPC в этой очереди после завершения функции обратного вызова *EvtInterruptIsr* и текущей процедуры DPC.

Но на многопроцессорных системах на другом процессоре могло бы произойти второе прерывание и эти две процедуры DPC, или функция *EvtInterruptIsr* и процедура DPC, могли бы исполняться одновременно. Например, допустим, что устройство выдает прерывание на процессоре 1, в то время как его функция *EvtInterruptDpc* исполняется на процессоре 0. Для обработки этого прерывания система исполняет функцию *EvtInterruptIsr* на процессоре 1. Когда функция *EvtInterruptIsr* ставит в очередь свою функцию *EvtInterruptDpc*, система ставит объект DPC в очередь процедур DPC процессора 1. Таким образом, функция *EvtInterruptIsr* драйвера может исполняться одновременно с его функцией DPC, и эта же функция DPC может исполняться одновременно на двух процессорах. Попытка обеими функциями одновременно получить доступ к разделяемым данным может породить серьез-

ные ошибки. Для защиты разделяемых данных при таких сценариях драйверы должны применять спин-блокировки для объекта прерываний, вызывая метод `WdfInterruptAcquireLock` или `WdfInterruptSynchronize`.

## Уровень IRQL DIRQL

Уровень IRQL DIRQL описывает диапазон значений IRQL, которые могут выдавать физические устройства. Как было показано ранее в табл. 15.1, каждая процессорная архитектура имеет диапазон значений уровней IRQL. Один и тот же уровень IRQL может использоваться несколькими устройствами. Уровень IRQL каждого устройства доступный его драйверу в списке преобразованных ресурсов (*translated resource list*), который инфраструктура передает функции обратного вызова `EvtDevicePrepareHardware` драйвера.

На уровне IRQL исполняются следующие функции обратного вызова драйверов KMDF:

- ◆ `EvtInterruptIsr`;
- ◆ `EvtInterruptSynchronize`;
- ◆ `EvtInterruptEnable` и `EvtInterruptDisable`.

Эти функции обратного вызова обычно содержатся в функциональных драйверах для физических устройств, генерирующих прерывания. В драйверах фильтров они встречаются редко.

При исполнении на уровне IRQL код драйвера должен соответствовать требованиям, описанных в разд. "Рекомендации по исполнению на уровне IRQL DISPATCH\_LEVEL или высшем" далее в этой главе.

### Полезная информация

Компания Microsoft внесла несколько улучшений в структуру прерываний Windows Vista, с которыми можно ознакомиться в статье **Interrupt Architecture Enhancements in Windows** (Улучшения в архитектуре прерываний Windows) по адресу <http://go.microsoft.com/fwlink/?LinkId=81584>.

## Уровень IRQL HIGH\_LEVEL

На уровне  $IRQL = HIGH\_LEVEL$  исполняются определенные функции `bugcheck` и функции обратного вызова для немаскируемых прерываний (NMI, Nonmaskable interrupt). Так как на уровне  $IRQL = HIGH\_LEVEL$  прерывания не допускаются, этим функциям гарантируется исполнение без прерываний.

Однако отсутствие прерываний означает, что функции обратного вызова очень ограничены в своих действиях. Коду, исполняющемуся на уровне  $HIGH\_LEVEL$ , запрещается выделять память, использовать любые механизмы синхронизации или вызывать любые функции, исполняющиеся на уровне  $IRQL \leq DISPATCH\_LEVEL$ . Имеются и другие ограничения, которые описываются в следующем разделе.

Информацию о написании функций NMI и `bugcheck` см. в разделе **Writing a Bug Check Callback Routine** (Написание процедуры обратного вызова для останова `bugcheck`) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81587>. Также см. раздел **KeRegisterNmiCallback** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81588>.

## Рекомендации по исполнению на уровне IRQL DISPATCH\_LEVEL или высшем

При исполнении кода драйвера на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$  необходимо придерживаться следующих руководящих принципов.

- ◆ Применяйте только нестраничные данные и код; не выполняйте никаких операций, требующих применения файла подкачки.

Windows должна ожидать завершения операций ввода/вывода с файлом подкачки, а это го нельзя делать на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ . По этой же причине, любые драйверные функции, захватывающие спин-блокировку, должны быть нестраничными.

Драйвер может хранить данные, к которым он будет обращаться при исполнении на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , в следующих местах:

- в области контекста объекта устройства, объекта процедуры DPC или любого другого объекта, передаваемого функции обратного вызова;
- в стеке ядра, для хранения небольших объемов данных, которые нет надобности хранить дольше, чем время жизни функции;
- в нестраничной памяти, выделяемой драйвером. Для больших объемов данных, таких как буферы ввода/вывода, драйверы должны создавать объекты памяти WDF, вызывая метод `WdfMemoryCreate`, или должны вызывать функции семейства `ExAllocateXxx` или `MmAllocateXxx`, в зависимости от того, что является уместным.

- ◆ Период ожидания на wait-блокировку WDF или объект диспетчера ядра — такой как объект события, семафора, таймера, мьютекса ядра, потока, процесса или файла — никогда не должен быть больше нуля.
- ◆ Не вызывайте функции, которые преобразуют данные из ANSI в Unicode и наоборот, т. к. эти функции содержат страничный код. Методы семейства `WdfStringXxx` и строковые функции, которые можно безопасно выполнять в режиме ядра, можно вызывать только на уровне  $\text{IRQL} = \text{PASSIVE\_LEVEL}$ .
- ◆ Никогда не вызывайте метод `WdfSpinLockRelease`, не вызвав предварительно метод `WdfSpinLockAcquire`.

## Вызовы функций, исполняющихся на низких уровнях IRQL

Если функция, исполняющаяся на высоком уровне IRQL, должна выполнить какую-либо длительную обработку, она организует завершение этой обработки на низшем уровне IRQL. Например, т. к. обратные вызовы функции `EvtInterruptIsr` исполняются на уровне DIRQL, они должны выполнять минимальный объем работы, поэтому они ставят в очередь процедуру DPC для завершения обработки на уровне  $\text{IRQL} = \text{DISPATCH\_LEVEL}$ .

Иногда код драйвера, исполняющийся на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , должен взаимодействовать с кодом с более низким уровнем IRQL. Например, если во время завершения операции ввода/вывода произойдет ошибка, драйверу USB может потребоваться выполнить сброс своего устройства. Функции обратного вызова `CompletionRoutine` могут вызываться на уровне  $\text{DISPATCH\_LEVEL}$ , но синхронные методы для выполнения сброса канала USB и устройства должны вызываться на уровне  $\text{PASSIVE\_LEVEL}$ . В такой ситуации драйвер мо-

жет использовать рабочий элемент. Код, вызывающий метод для выполнения сброса, находится в функции обратного вызова `EvtWorkItem`. Драйвер создает объект рабочего элемента WDF, ассоциированного с функцией, после чего ставит рабочий элемент в очередь. Инфраструктура добавляет функцию рабочего элемента в системную очередь рабочих элементов, которую система исполняет позже в контексте системного потока на уровне `IRQL = PASSIVE_LEVEL`. Кроме этого, если драйвер устанавливает уровень исполнения для функций обратного вызова для объекта файла и события ввода/вывода в `WdfExecutionLevelPassive`, то инфраструктура вызывает эти функции из рабочего элемента.

Дополнительную информацию по этому вопросу см. в разд. "Рабочие элементы и драйверные потоки" далее в этой главе.

## Сценарии прерывания потоков

Может ли исполняющийся поток быть прерванным или вытесненным, определяется приоритетом планирования потока и текущим уровнем IRQL процессора. При вытеснении потока операционная система заменяет поток, исполняющийся на данном процессоре, другим потоком, обычно имеющим высший приоритет потока. Эффектом, оказываемым вытеснением на отдельный поток, является недоступность этому потоку процессора на некоторое время. При прерывании потока операционная система заставляет текущий поток временно исполнять код на более высоком уровне IRQL.

Понимание происходящего при вытеснении или прерывании потока можно облегчить, рассмотрев несколько простых примеров. Такие примеры приводятся в следующем разделе: один для однопроцессорной системы, а другой — для многопроцессорной.

## Прерывание потока на однопроцессорной системе

На рис. 15.1 показана гипотетическая ситуация выполнения прерывания потока на однопроцессорной системе. Для простоты, в примере опущены планировщик потока, прерывания по таймеру и прочие несущественные компоненты.

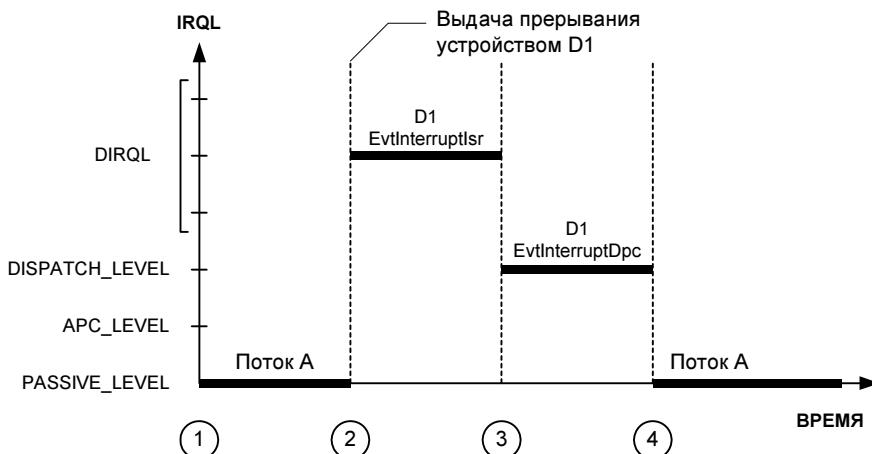


Рис. 15.1. Прерывание потока на однопроцессорной системе

Различные этапы прерывания потока, всего четыре, обозначены на рисунке цифрами. Процесс протекает следующим образом:

1. Поток А исполняется на уровне IRQL = PASSIVE\_LEVEL.
2. Устройство 1 выдает прерывание на уровне DIRQL. Поток А прерывается, хотя выделенный ему временной квант еще не истек. Система приостанавливает исполнение потока А и исполняет процедуру обратного вызова для устройства 1. Функция `EvtInterruptIsr` прекращает прерывание устройством 1, сохраняет данные, которые ей могут потребоваться, регистрирует функцию `EvtInterruptDpc` и возвращает управление.
3. Прерываний с уровнем IRQL больше нет ни для каких устройств. Так как процедуры DPC исполняются на уровне IRQL = DISPATCH\_LEVEL, прежде чем поток А может возобновить свою работу, должны быть исполнены все члены системной очереди процедур DPC.

В данном случае стоящая в очереди процедура DPC исполняет функцию `EvtInterruptDpc`, зарегистрированную функцией `EvtInterruptIsr` устройства 1. Так как на уровне  $\text{IRQL} > \text{DISPATCH\_LEVEL}$  не происходит больше никаких прерываний, то функция `EvtInterruptDpc` исполняется полностью до своего завершения.

4. После возвращения управления функцией `EvtInterruptDpc` очередь процедур DPC становится пустой; также нет готовых к исполнению никаких других потоков с более высоким приоритетом. Поэтому система возобновляет исполнение потока А и продолжает его до тех пор, пока не произойдет одно из следующих событий:
  - выделенный потоку квант времени истек;
  - поток переходит в состояние ожидания;
  - поток завершает исполнение;
  - становится готовым к исполнению поток с более высоким приоритетом;
  - происходит аппаратное прерывание;
  - поток ставит в очередь процедуру DPC или APC.

Нет никакой гарантии, что поток А полностью исчерпает выделенный ему квант времени. В течение своего кванта времени поток может быть прерван или вытеснен неограниченное количество раз.

## Прерывание потока на многопроцессорной системе

На рис. 15.2 показана гипотетическая ситуация выполнения прерывания потока на многопроцессорной системе. Для простоты, в примере опущены прерывания по таймеру и прочие нерелевантные компоненты, но показан системный планировщик потоков.

Исполнение потоков на обоих процессорах начинается одновременно, а процесс прерывания протекает следующим образом:

1. Процессор 0 исполняет поток А, а процессор 1 исполняет поток В. Оба потока исполняются на уровне IRQL = PASSIVE\_LEVEL.
2. Устройство 1 выдает прерывание на процессоре 0, в результате чего система повышает IRQL для устройства 1 на процессоре 0 до уровня IRQL и исполняет функцию `EvtInterruptIsr` устройства 1. Функция `EvtInterruptIsr` устройства 1 ставит в очередь функцию `EvtInterruptDpc`. По умолчанию функция `EvtInterruptDpc` ставится в очередь

- того же самого процессора, на котором исполняется функция *EvtInterruptIsr* (т. е. в очередь процессора 0).
- Так как функция *EvtInterruptDpc* исполняется на том же самом процессоре, что и функция *EvtInterruptIsr*, но на более низком уровне IRQL, она не начинает исполняться до тех пор, пока функция *EvtInterruptIsr* не возвратит управление.

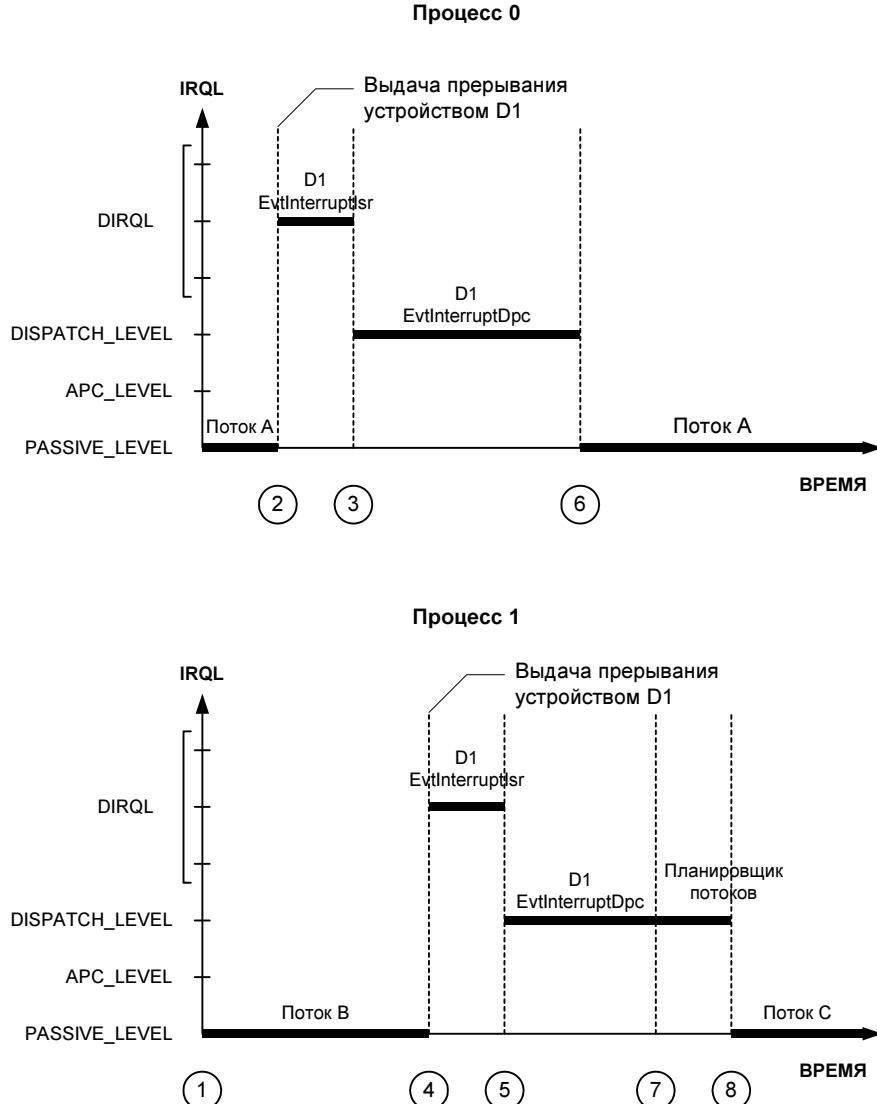


Рис. 15.2. Прерывание потока на многопроцессорной системе

- Устройство 1 снова выдает прерывание, в этот раз на процессоре 1. В результате этого прерывания система повышает уровень IRQL для устройства 1 на процессоре 1 до уровня DIRQL и исполняет функцию *EvtInterruptIsr* устройства 1.

5. Функция *EvtInterruptIsr* устройства 1 ставит в очередь на процессоре 1 функцию *EvtInterruptDpc* (что является действием по умолчанию). Функция *EvtInterruptDpc* начинает исполняться на процессоре 1 после возвращения управления функцией *EvtInterruptIsr*.
6. Теперь та же самая функция *EvtInterruptDpc* исполняется одновременно на обоих процессорах в ответ на два разных прерывания от одного и того же устройства. Чтобы обеспечить целостность разделяемых данных, к которым могут обращаться эти функции, драйвер должен применять спин-блокировки. В данном случае функция *EvtInterruptDpc* на процессоре 0 захватывает блокировку первой, поэтому процессор 1 "кружится" (spinning) в ожидании блокировки.
7. Функция *EvtInterruptDpc* на процессоре 0 освобождает блокировку, завершает свою работу и возвращает управление. Система теперь готова к исполнению следующего потока на процессоре 0, которым в данном случае является поток А.
8. После того как функция *EvtInterruptDpc* на процессоре 0 освободит блокировку, блокировку захватывает функция *EvtInterruptDpc* на процессоре 1 и выполняет свои задачи. После того как эта функция возвратит управление, исполняется код системного планировщика потоков, чтобы определить, какой поток исполнять следующим.
9. После возвращения управления планировщиком потоков в очередь не было поставлено никаких других процедур DPC, поэтому на процессоре 1 исполняется поток с самым высоким приоритетом, т. е. поток С.

## Тестирование на наличие проблем, связанных с IRQL

Проблемы, связанные с уровнями IRQL, довольно распространенные и часто вызывают системный сбой с кодом `bugcheck IRQL_NOT_LESS_OR_EQUAL`. KMDF выполняет определенную внутреннюю проверку правильности IRQL. Кроме этого, набор WDK содержит несколько возможностей, с помощью которых можно определить уровень IRQL, на котором исполняется код, и обнаружить проблемы, связанные с IRQL. В этом разделе описываются следующие из этих возможностей:

- ◆ функции и команды отладчика, возвращающие текущий уровень IRQL;
- ◆ макросы `PAGED_CODE` и `PAGED_CODE_LOCKED`;
- ◆ принудительная проверка IRQL в инструменте Driver Verifier.

Инструменты PREfast for Drivers и SDV также содержат возможности, которые можно применить для обнаружения проблем, связанных с IRQL. Подробная информация о применении этих инструментов для тестирования драйверов KMDF приводится в главах 23 и 24.

## Способы получения текущего уровня IRQL

Узнать текущий уровень IRQL, на котором работает процессор, можно одним из следующих способов:

- ◆ вызвав функцию `KeGetCurrentIrql`;
- ◆ с помощью команды `!irql` расширения отладчика режима ядра.

Драйвер может вызвать функцию `KeGetCurrentIrql` на любом уровне IRQL. Определить, работает ли он в критической области, драйвер может, вызывая функцию `KeAreApcDisabled`, которая доступна в Windows XP и более поздних выпусках.

При отладке определить уровень IRQL, на котором работает процессор, можно с помощью команды `!irql` расширения отладчика. Эта команда возвращает уровень IRQL, на котором процессор работал непосредственно перед тем, как был активирован отладчик. По умолчанию команда возвращает уровень IRQL для текущего процессора, но в параметре можно указать требуемый процессор. Эта команда работает на системах под управлением Windows Server 2003 и более поздних версиях Windows.

### **Макросы `PAGED_CODE` и `PAGED_CODE_LOCKED`**

Макросы `PAGED_CODE` и `PAGED_CODE_LOCKED` предназначены для обнаружения проблем с IRQL, связанных со страничными ошибками.

Если при вызове макроса `PAGED_CODE` процессор работает на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , система прерывает исполнение и запускает отладчик. Поместив макрос в начале каждой драйверной функции, содержащей или вызывающей страничный код, можно определить, выполняет ли функция действия, недопустимые на уровне IRQL, на котором она вызывается.

При совместном применении со статическими инструментами верификации, макрос `PAGED_CODE` может быть полезным в обнаружении проблем, связанных с IRQL, в коде, который впоследствии повышает уровень IRQL. Например, если при исполнении макроса `PAGED_CODE` процессор работает на уровне `PASSIVE_LEVEL`, но функция впоследствии вызывает метод `WdfSpinLockAcquire` — который повышает уровень IRQL до `DISPATCH_LEVEL` — то система выдаст предупреждение.

Дополнительную информацию по этому вопросу см. в разделе **PAGED\_CODE** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82722>.

Некоторые функции могут содержать страничный код или данные, но для правильного исполнения должны блокироваться в памяти. Поместив макрос `PAGED_CODE_LOCKED` в начале такой функции, можно обеспечить ее блокировку при вызове. Если код, вызывающий этот макрос, не заблокирован в памяти, система прерывает штатное исполнение и запускает отладчик.

Дополнительную информацию по этому вопросу см. в разделе **PAGED\_CODE\_LOCKED** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=82723>.

### **Опции инструмента Driver Verifier**

Инструмент Driver Verifier (`Verifier.exe`) выполняет множественные проверки, связанные с IRQL. По умолчанию Driver Verifier проверяет на наличие определенных ошибок, связанных с повышением и понижением уровня IRQL, выделением памяти на недопустимом уровне IRQL и захватом и освобождением спин-блокировок на недопустимых уровнях IRQL.

Кроме этого, Driver Verifier может выполнять принудительную проверку уровня IRQL. При задействовании этой опции Driver Verifier помечает весь страничный код и данные, когда драйвер или инфраструктура запрашивает спин-блокировку, исполняет функцию обратного вызова `EvtInterruptSynchronize` или повышает уровень IRQL до `DISPATCH_LEVEL` или выше. При попытке драйвера обратиться к любой области страничной памяти Driver Verifier выдает останов `bugcheck`.

При задействованной принудительной проверке IRQL Driver Verifier собирает статистические данные, связанные с IRQL, включая количество раз, когда драйвер повышал уровень IRQL, захватывал спин-блокировку или вызывал процедуру `KeSynchronizeExecution`, которая

является функцией режима ядра, лежащей в основе метода `WdfInterruptSynchronize`. Driver Verifier также ведет счет, сколько раз операционная система сбрасывала страничную память на диск. Все эти статистические данные Driver Verifier хранит в глобальных счетчиках. Значения этих счетчиков можно вывести на экран в консольном или графическом интерфейсе Driver Verifier или с помощью расширения `!verifier` в отладчике. Для драйверов, выполняющих операции DMA, также следует использовать опцию Driver Verifier для верификации операций DMA. Эта опция проверяет на наличие вызовов на неправильном уровне IRQL функций, выполняющих операции DMA.

### Полезная информация

Синтаксис команд Driver Verifier зависит от установленной версии Windows. Дополнительную информацию о Driver Verifier см. в главе 21.

## Рабочие элементы и драйверные потоки

Хотя драйверы KMDF могут создавать новый поток, вызывая метод `PsCreateSystemThread`, они редко это делают. Переключение контекста потока забирает сравнительно много времени и частое переключение потоков может понизить производительность драйвера. Поэтому драйверы должны создавать выделенные потоки только для исполнения постоянно повторяющихся или длительных операций, таких как, например, опрашивание (polling) устройства или управление несколькими потоками данных, как может быть в случае с сетевым драйвером.

Для выполнения кратковременных, конечных операций драйвер не должен создавать собственные потоки. Вместо этого, он может "одолжить" на время системный поток, поставив в очередь рабочий элемент. Система содержит выделенный пул потоков, совместно используемых всеми драйверами. Когда драйвер ставит в очередь рабочий элемент, система отправляет его для исполнения в один из этих общих потоков. Драйверы применяют рабочие элементы для исполнения кода в системном потоке и контексте безопасности или для вызова функций, доступных только на уровне `IRQL = PASSIVE_LEVEL`. Функция обратного вызова `CompletionRoutine` драйвера (которую инфраструктура может вызывать на уровне `IRQL = DISPATCH_LEVEL`) часто использует рабочие элементы для обращения к страничным данным или для вызова функций, выполняющихся на уровне `IRQL = PASSIVE_LEVEL`.

Так как система имеет ограниченное количество выделенных рабочих потоков, назначаемые им задания должны выполняться в быстром порядке. При реализации рабочих элементов в драйвере стоит учесть следующие рекомендации.

- ◆ Не создавайте рабочих элементов, исполняющихся непрерывно до выгрузки драйвера. Вместо этого, ставьте рабочие элементы в очередь, только когда требуется. По завершению своей задачи функция рабочего элемента должна возвращать управление.
- ◆ Никогда не вставляйте бесконечных циклов в рабочие элементы.
- ◆ Следует избегать слишком большого количества рабочих элементов, т. к. занимание системных рабочих потоков может заблокировать систему. Вместо этого создайте одну функцию рабочего элемента, которая выполняет всю имеющуюся работу и возвращает управление, когда больше нет никаких задач, требующих немедленного исполнения.
- ◆ Не ожидайте события в течение длительного времени ни в каком рабочем элементе. В особенности не ожидайте события, подаваемого другим рабочим элементом. Если все рабочие потоки заняты, система не планирует на исполнение новый рабочий элемент до

тех пор, пока какой-либо рабочий элемент не завершит свою работу. В такой ситуации существует возможность возникновения взаимоблокировки.

## Рабочие элементы

В драйверах KMDF рабочий элемент представляется объектом `WDFWORKITEM`. Для работы с рабочим элементом драйвер должен выполнить следующие действия:

- ◆ реализовать функцию обратного вызова для рабочего элемента;
- ◆ сконфигурировать и создать объект рабочего элемента;
- ◆ поставить объект рабочего элемента в очередь.

Функция обратного вызова для рабочего элемента имеет следующий прототип:

```
typedef VOID
(*PFN_WDF_WORKITEM) (
    IN WDFWORKITEM WorkItem
);
```

Как можно видеть в прототипе, функция обратного вызова `EvtWorkItem` не возвращает значения. Ей передается лишь один параметр — сам объект `WDFWORKITEM`. Для передачи данных функции обратного вызова `EvtWorkItem` драйвер должен использовать область контекста объекта рабочего элемента. Эта функция обратного вызова выполняет задачи, поставленные рабочему элементу.

Каждый рабочий элемент ассоциирован с определенной функцией обратного вызова `EvtWorkItem`. Когда драйвер вызывает метод `WdfWorkItemEnqueue`, то инфраструктура добавляет рабочий элемент в системную очередь отложенных заданий.

Позже системный рабочий поток забирает рабочий элемент из очереди и исполняет его функцию обратного вызова `EvtWorkItem` в контексте системного потока на уровне `IRQL = PASSIVE_LEVEL`. Для синхронизации действий функции обратного вызова с действиями других функций драйвера, драйвер может применить `wait-блокировку` WDF или механизм синхронизации, предоставляемый Windows.

## Пример KMDF: применение рабочего элемента

Образец драйвера `Usbsamp` выполняет ввод/вывод в канал получателя USB. При определенных ошибках в процессе завершения ввода/вывода в своей функции обратного вызова завершения ввода/вывода драйвер ставит в очередь рабочий элемент, чтобы выполнить сброс канала получателя. Драйвер не может выполнить сброс канала непосредственно из функции обратного вызова завершения ввода/вывода. Причина в том, что инфраструктура может активизировать функцию обратного вызова завершения ввода/вывода на уровне `DISPATCH_LEVEL`, а метод `WdfUsbTargetPipeResetSynchronously`, вызываемый драйвером для выполнения сброса канала, необходимо вызывать на уровне `PASSIVE_LEVEL`.

В листинге 15.1 показан код из файла `Usbsamp\Sys\Bulkwr.c`, в котором создается и ставится в очередь рабочий элемент.

### Листинг 15.1. Создание и постановка в очередь рабочего элемента

```
NTSTATUS QueuePassiveLevelCallback(
    IN WDFDEVICE Device,
    IN WDFUSBPIPE Pipe)
```

```

{
    NTSTATUS status = STATUS_SUCCESS;
    PWORKITEM_CONTEXT context;
    WDF_OBJECT_ATTRIBUTES attributes;
    WDF_WORKITEM_CONFIG workitemConfig;
    WDFWORKITEM hWorkItem;
    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE(&attributes, WORKITEM_CONTEXT);
    attributes.ParentObject = Device;
    WDF_WORKITEM_CONFIG_INIT(&workitemConfig, ReadWriteWorkItem);
    status = WdfWorkItemCreate(&workitemConfig,
        &attributes,
        &hWorkItem);

    if (!NT_SUCCESS(status)) {
        return status;
    }

    context = GetWorkItemContext(hWorkItem);
    context->Device = Device;
    context->Pipe = Pipe;
    WdfWorkItemEnqueue(hWorkItem);
    return STATUS_SUCCESS;
}

```

Прежде чем создать объект рабочего элемента, драйвер, приведенный в листинге 15.1, конфигурирует структуру атрибутов и структуру конфигурации рабочего элемента. Инициализация структуры атрибутов выполняется функцией `WDF_OBJECT_ATTRIBUTES_INIT`, а тип области контекста объекта устанавливается макросом `WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE`. Драйвер использует область контекста, чтобы передавать данные функции обратного вызова рабочего элемента.

По умолчанию у объекта рабочего элемента нет родителя, поэтому драйвер `Usbsamp` указывает в качестве родителя объект устройства, устанавливая поле `Parent` структуры атрибутов. Установка объекта устройства в качестве родителя обеспечивает, что инфраструктура удалит объект рабочего элемента при удалении объекта устройства.

Драйвер `Usbsamp` инициализирует структуру конфигурации рабочего элемента, вызывая функцию `WDF_WORKITEM_CONFIG_INIT`, передавая ей в качестве параметров указатель на структуру конфигурации рабочего элемента и указатель на функцию обратного вызова `EvtWorkItem` драйвера под названием `ReadWriteWorkItem`.

После этого драйвер `Usbsamp` создает объект рабочего элемента, вызывая метод `WdfWorkItemCreate` и передавая ему указатели на только что инициализированные структуры. Метод `WdfWorkItemCreate` возвращает дескриптор объекта рабочего элемента.

При условии, что создание объекта рабочего элемента прошло успешно, драйвер `Usbsamp` получает указатель на область контекста рабочего элемента, вызывая функцию доступа `GetWorkItemContext`. Драйвер сохраняет дескрипторы объекта устройства и объекта канала, переданные функции `QueuePassiveLevelCallback`, в области контекста, так что при выполнении рабочего элемента он может иметь незатрудненный доступ к ним. После этого драйвер вызывает метод `WdfWorkItemEnqueue`, передавая ему только что созданный им объект рабочего элемента.

Хотя образец драйвера создает и ставит в очередь рабочий элемент во время процесса завершения ввода/вывода, драйверу, который часто ставит в очереди такие рабочие элементы,

следует вместо этого использовать предварительно выделенный объект рабочего элемента. Потом драйвер может повторно использовать этот объект, таким образом избегая любых возможных ошибок с выделением, которые могли бы возникнуть в функции обратного вызова завершения ввода/вывода. Но драйвер не должен повторно ставить в очередь один и тот же рабочий элемент до тех пор, пока не завершится исполнение функции обратного вызова ранее поставленного в очередь рабочего элемента. В образце драйвера показано, каким образом можно отслеживать состояние рабочего элемента в таких ситуациях, используя флаг и работу со взаимной блокировкой.

В листинге 15.2 приведен исходный код функции обратного вызова *EvtWorkItem* образца драйвера UsbSamp. Эта функция также определена в исходном файле Usbsamp\Sys\Bulkwr.c.

#### Листинг 15.2. Функция обратного вызова для рабочего элемента

```
VOID ReadWriteWorkItem(IN WDFWORKITEM WorkItem)
{
    PWORKITEM_CONTEXT pItemContext;
    NTSTATUS status;
    pItemContext = GetWorkItemContext(WorkItem);
    status = ResetPipe(pItemContext->Pipe);
    if (!NT_SUCCESS(status)) {
        status = ResetDevice(pItemContext->Device);
        if (!NT_SUCCESS(status)){
            . . . // Код опущен для краткости.
        }
    }
    WdfObjectDelete(WorkItem);
    return;
}
```

Функция обратного вызова для рабочего элемента выполняет сброс канала получателя и, если необходимо, порта USB. При вызове этой функции в качестве параметра ей передается дескриптор рабочего элемента. Драйвер немедленно вызывает функцию доступа, чтобы получить указатель на область контекста рабочего элемента, после чего передает дескриптор канала, сохраненный ранее в области контекста, вспомогательной функции *ResetPipe*. Наконец, для выполнения сброса канала USB функция *ResetPipe* вызывает инфраструктурный метод *WdfUsbTargetPipeResetSynchronously*. Если эта операция завершается неудачей, драйвер вызывает свою вспомогательную функцию *ResetDevice*, которая, в свою очередь, вызывает метод *WdfUsbTargetDeviceResetPortSynchronously* для выполнения сброса порта USB.

По завершению выполнения задач сброса, в объекте рабочего элемента больше нет надобности. Драйвер удаляет его, вызывая метод *WdfObjectDelete*, после чего функция обратного вызова возвращает управление.

## Оптимальные методики для управления контекстом потока и IRQL в драйверах KMDF

Чтобы избежать проблем, связанных с контекстом потока и уровнями IRQL, придерживайтесь следующих правил.

- ◆ Если у вас нет полной уверенности в том, что драйверная функция вызывается в контексте определенного потока, никогда не делайте никаких предположений относительно содержания адресного пространства пользовательского режима.
- ◆ Необходимо знать, какие драйверные функции можно вызывать на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$  и понимать ограничения, накладываемые на код драйвера исполнением на этом уровне.
- ◆ Все данные, к которым можно получить доступ на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , необходимо хранить в нестраничной памяти. Например, в области контекста объекта устройства, в выделенной драйвером области памяти нестраничного пула или в стеке режима ядра.
- ◆ Тестируйте драйверы на ошибки, связанные с IRQL, с помощью Driver Verifier, макросов PAGED\_CODE и PAGED\_CODE\_LOCKED, Static Driver Verifier, PRefast for Drivers и расширений отладчика.
- ◆ Выполняйте тестирование драйверов на как можно большем числе конфигураций, включая многопроцессорные системы.

# ГЛАВА 16

## Аппаратные ресурсы и прерывания

Устройства, подключаемые к шине PCI, шине PCI Express или другой монтажнойшине, обычно генерируют прерывания, чтобы дать знать процессору — и, таким образом, операционной системе — что они нуждаются в обслуживании. Когда пользователь подключает такое устройство к системе, менеджер PnP Windows, с учетом информации, предоставляемой ему драйвером устройства, выделяет ему набор аппаратных ресурсов. Среди прочих, выделяются такие ресурсы, как отображенное на память пространство ввода/вывода, векторы прерываний, каналы DMA и другие ресурсы, с помощью которых устройство взаимодействует с системой. В зависимости от типа устройства и типов ресурсов, драйверу может потребоваться программный код для управления этими ресурсами. Если устройство генерирует прерывания, то его функциональный драйвер должен содержать программный код для обработки этих прерываний.

В этой главе описывается, каким образом драйвер отображает ресурсы памяти на виртуальную память, с тем, чтобы он мог обращаться к ним. Также объясняется, что должен делать функциональный драйвер для обработки прерываний.

Ресурсы, необходимые для данной главы	Расположение
<b>Образцы драйверов</b>	
Pcidrv	%wdk%\Src\Kmdf\Pcidrv
PLx9x5x	%wdk%\Src\Kmdf\PLX9x5x
<b>Документация WDK</b>	
Процедура MmMapIoSpace	<a href="http://go.microsoft.com/fwlink/?LinkId=81589">http://go.microsoft.com/fwlink/?LinkId=81589</a>
Процедуры уровня абстрагирования от оборудования	<a href="http://go.microsoft.com/fwlink/?LinkId=81591">http://go.microsoft.com/fwlink/?LinkId=81591</a>

## Аппаратные ресурсы

Когда Windows обнаруживает подключенное к системе устройство, драйвер этого устройства должен выполнить обработку аппаратных ресурсов этого устройства. Драйвер определяет порты ввода/вывода, отображенные на память адреса и прерывания, применяемые для взаимодействия с устройством. Эту информацию драйвер сохраняет в определяемой драйвером

области памяти для дальнейшего использования и отображает ресурсы, основанные на памяти, на виртуальное адресное пространство режима ядра.

В зависимости от типа устройства, типа шины, к которому оно подключено, и базовой аппаратной платформы, регистры устройства можно отобразить на память или на системное пространство ввода/вывода. Большинство современных процессоров и распространенных шин — включая шины PIC, PCI Express, ISA и EISA — поддерживают отображение, как на пространство памяти, так и на пространство ввода/вывода.

Отображение на пространство ввода/вывода является наследием архитектур ранних микропроцессоров, когда редко какое устройство имело собственную адресуемую память. Пространство ввода/вывода было разработано как отдельное адресное пространство для обращения к регистрам устройств, таким образом позволяя сэкономить дефицитное адресное пространство для использования операционной системой и приложениями. Но в настоящее время регистры устройств обычно отображаются на адресное пространство памяти. В устройствах с отображением на память регистры устройства отображаются на адресное пространство физической памяти. Перед использованием этих адресов драйвер отображает их на виртуальное адресное пространство следующим образом.

- ◆ Регистры устройства, отображаемые на пространство ввода/вывода, считаются и записываются с помощью макросов семейств READ\_PORT\_Xxx и WRITE\_PORT\_Xxx.  
Поэтому отображение на пространство ввода/вывода иногда называется *отображением на порты* (PORT<sup>1</sup> mapping), а отображаемые ресурсы — *портовыми ресурсами* (PORT resources).
- ◆ Регистры устройства, отображаемые на пространство памяти, считаются и записываются с помощью макросов семейства READ\_REGISTER\_Xxx и WRITE\_REGISTER\_Xxx.

Отображение на память иногда называется *регистровым отображением* (REGISTER mapping), а соответствующие ресурсы — *регистровыми ресурсами* (REGISTER resources).

Хотя вы можете знать конструкцию аппаратной части вашего устройства, эта информация не обязательно будет полезной для определения, каким образом будут отображены его ресурсы, т. к. некоторые чипсеты изменяют тип отображения. Поэтому драйверы должны обеспечивать отображение обоих типов для каждого регистра. Для обеспечения поддержки обоих типов отображения широко применяется способ, при котором определяются оберточные интерфейсы (wrappers) для макросов PORT и REGISTER.

## Идентификация и освобождение аппаратных ресурсов

Функциональный драйвер идентифицирует и освобождает аппаратные ресурсы, требуемые его устройству, в функциях обратного вызова по событию *EvtDevicePrepareHardware* и *EvtDeviceReleaseHardware* соответственно. Эти функции обратного вызова предоставляют драйверу способ для подготовки аппаратного обеспечения своего устройства при физическом удалении или подключении или при его остановке для перераспределения ресурсов. Инфраструктура вызывает функцию *EvtDevicePrepareHardware* непосредственно перед вызовом функции *EvtDeviceD0Entry*, а функцию *EvtDeviceReleaseHardware* — сразу же после возвращения управления функцией *EvtDeviceD0Exit*.

<sup>1</sup> В оригинальной, английской, терминологии слово PORT может писаться или заглавными, или строчными буквами. — *Пер.*

Подробное объяснение последовательности обратных вызовов функций, принимающих участие в запуске и остановке устройства, приводится в главе 7.

## Идентификация ресурсов: подготовка аппаратного обеспечения

При выделении ресурсов устройству инфраструктура вызывает функцию *EvtDevicePrepareHardware*, особенно в следующих случаях:

- ◆ при первоначальном запуске системы;
- ◆ при подключении пользователем устройства к работающей системе;
- ◆ при перезапуске устройства после его остановки для перераспределения ресурсов.

Функция *EvtDevicePrepareHardware* должна отобразить ресурсы драйвера, но не должна загружать микропрограммное обеспечение (firmware) или выполнять иные задачи инициализации. Инфраструктура вызывает функцию *EvtDevicePrepareHardware* только при первоначальном запуске системы или устройства или после остановки устройства для перераспределения ресурсов. Функция *EvtDevicePrepareHardware* не вызывается после возвращения устройства в рабочее состояние из состояния пониженного энергопотребления. Если при переходе в состояние пониженного энергопотребления очищается загруженное микропрограммное обеспечение, то для его восстановления функция *EvtDevicePrepareHardware* не вызывается.

В типичном драйвере эта функция обратного вызова сохраняет копии ресурсов в области контекста устройства для дальнейшего использования и отражает физические адреса всех ресурсов памяти на виртуальные адреса ядра.

При вызове инфраструктурой функции *EvtDevicePrepareHardware*, хотя устройство уже адресуемо, оно еще не находится в рабочем состоянии. Драйвер должен ограничиться в своем обращении к аппаратному обеспечению только действиями, необходимыми для правильной идентификации версии устройства.

## Освобождение аппаратного обеспечения

Функция обратного вызова *EvtDeviceReleaseHardware* отменяет все действия, выполненные функцией обратного вызова *EvtDevicePrepareHardware*. Инфраструктура вызывает функцию *EvtDeviceReleaseHardware* при следующих обстоятельствах:

- ◆ при остановке системы, но не при остановке отдельного устройства;
- ◆ при удалении устройства из системы;
- ◆ при остановке устройства для перераспределения ресурсов.

В функции *EvtDeviceReleaseHardware* драйвер уничтожает все программное состояние, установленное им в функции *EvtDevicePrepareHardware*. Обычно эта функция отменяет отображение ресурсов, основанных на памяти. Драйвер не должен обращаться к аппаратному обеспечению из функции обратного вызова *EvtDeviceReleaseHardware*, т. к. аппаратные ресурсы драйвера уже были возвращены системе, а устройство уже перешло в состояние D3. К тому же если устройство было неожиданно удалено, то оно также отсутствует физически.

Драйвер регистрирует функции обратного вызова для событий *EvtDevicePrepareHardware* и *EvtDeviceReleaseHardware* в функции *EvtDriverDeviceAdd*. Чтобы зарегистрировать обратные вызовы этих функций, драйвер инициализирует соответствующие поля в структуре *Wdf\_PNPPOWER\_EVENT\_CALLBACKS*, после чего вызывает метод *WdfDeviceInitSetPnpPowerEvent-*

Callbacks, чтобы внести информацию в структуру `WDFDEVICE_INIT`, прежде чем создавать объект устройства WDF.

Дополнительную информацию о регистрации этих функций обратного вызова см. в главе 7.

## Списки ресурсов

При перечислении устройств, в ответ на запрос от менеджера PnP драйвер шины запрашивает отображение на пространство ввода/вывода или на пространство памяти. Менеджер PnP загружает объекты FDO для каждого устройства и повторяет запрос, чтобы функциональный драйвер мог добавить или удалить ресурсы со списка. Для участия в процессе запроса ресурсов функциональный драйвер KMDF может зарегистрировать функции обратных вызовов семейства `EvtDeviceFilterXxx`.

Менеджер PnP выделяет устройству "сырые" и преобразованные ресурсы и создает списки выделенных ресурсов. Инфраструктура получает эти списки ресурсов от менеджера PnP и передает их функциональному драйверу KMDF в функции обратного вызова `EvtDevicePrepareHardware`. Таким образом, драйвер получает два списка ресурсов.

- ◆ **Список "сырых" ресурсов.** В этом списке определены аппаратные ресурсы с точки зрения шины ввода/вывода, к которой подключено устройство. Список "сырых" ресурсов указывает конструкцию устройства.
- ◆ **Список преобразованных ресурсов.** В этом списке ресурсы определены с точки зрения шины процессора. В нем указывается отображение каждого ресурса в текущей системе и, таким образом, тип макроса — POST или REGISTER — который драйвер должен использовать для чтения и записи в ресурс.

Как список "сырых", так и список преобразованных ресурсов описывают те же самые ресурсы в том же самом порядке, при взаимно-однозначном соответствии между списками. Разница между этими двумя списками заключается в том, что в списке "сырых" ресурсов адреса указаны по отношению кшине устройства, а список преобразованных ресурсов содержит системные физические адреса.

## Анализ аппаратных ресурсов драйвером

Чтобы определить тип отображения для каждого аппаратного ресурса, драйвер анализирует список преобразованных ресурсов.

Каждый список ресурсов представлен объектом типа `WDFCMRESLIST` и описывает один или несколько ресурсов. Чтобы определить количество ресурсов, выделенных его устройству, драйвер вызывает метод `WdfCmResourceListGetCount`, передавая ему список преобразованных ресурсов. Потом драйвер может получить подробную информацию о каждом отдельном ресурсе, вызывая в цикле метод `WdfCmResourceListGetDescriptor` для каждого элемента списка.

Метод `WdfCmResourceListGetDescriptor` возвращает указатель на структуру `CM_PARTIAL_RESOURCE_DESCRIPTOR`, описывающую отдельный ресурс. Данная структура содержит следующие поля:

- ◆ `Type` — содержит константу, указывающую тип ресурса. Если вы знакомы с драйверами WDM, то заметите, что типы ресурсов (перечисленные в табл. 16.1) такие же, как и типы для драйверов WDM;
- ◆ `ShareDisposition` — указывает, является ли данный ресурс разделяемым;

- ◆ Flags — предоставляет флаги, специфичные для определенного типа;
- ◆ объединение u — предоставляет дополнительную информацию, специфичную для данного типа ресурса, как изложено в табл. 16.1.

**Таблица 16.1. Типы ресурсов и соответствующие члены объединения в дескрипторе ресурса**

Тип	Член объединения u для типа
CmResourceTypePort	u.Port
CmResourceTypeInterrupt	u.Interrupt для вектора прерываний или u.MessageInterrupt сообщения MSI.  Если в поле Flags установлен флаг CM_RESOURCE_INTERRUPT_MESSAGE, используйте u.MessageInterrupt; в противном случае — u.Interrupt
CmResourceTypeMemory	u.Memory
CmResourceTypeMemoryLarge	Один из: u.Memory40, u.Memory48 или u.Memory64.  Флаги CM_RESOURCE_MEMORY_LARGE_Xxx, установленные в поле Flags, указывают используемую структуру
CmResourceTypeDma	u.Dma
CmResourceTypeDevicePrivate	u.DevicePrivate
CmResourceTypeBusNumber	u.BusNumber
CmResourceTypeDeviceSpecific	u.DeviceSpecificData
CmResourceTypePcCardConfig	u.DevicePrivate
CmResourceTypeMfCardConfig	u.DevicePrivate
CmResourceTypeConfigData	Зарезервировано для системного пользования
CmResourceTypeNonArbitrated	Не используется

Обычно драйверы могут игнорировать ресурсы любых типов, кроме типов CmResourceTypeMemory, CmResourceTypePort, CmResourceTypeInterrupt и CmResourceTypeMemoryLarge.

Ресурсы типов CmResourceTypeMemory и CmResourceTypeMemoryLarge отображаются на память, а ресурсы типа CmResourceTypePort — на пространство ввода/вывода. Для каждого ресурса указывается начальный адрес и размер. Драйверы должны проанализировать и сохранить списки преобразованных ресурсов, а потом использовать преобразованные ресурсы для чтения и записи в регистры устройства. Для большинства драйверов не требуется сырых ресурсов.

### Платформенная независимость и отображение ресурсов драйвера

Независимо от конструкции самого устройства, чипсет конкретной системы может отображать аппаратные ресурсы или на пространство ввода/вывода, или на пространство памяти. Чтобы не зависеть от платформы, все драйверы должны поддерживать оба типа отображения ресурсов.

- ◆ Для ресурса, отображаемого на пространство ввода/вывода (т. е. типа CmResourceTypePort), драйвер сохраняет базовый адрес и диапазон отображения ресурса. Драйвер также со-

храняет указатель на внутреннюю функцию, которая с помощью макроса `PORT` получает доступ к ресурсу.

- ◆ Для ресурса, отображаемого на пространство памяти (т. е. типа `CmResourceTypeMemory` или `CmResourceTypeMemoryLarge`), драйвер проверяет, является ли выделенный объем памяти достаточным. При положительном результате проверки драйвер отображает возвращенный физический адрес на виртуальный адрес, вызывая для этого процедуру `MmMapIoSpace`, и сохраняет указатели на внутренние функции, которые используют макросы `REGISTER` для доступа к ресурсу.

Процедура `MmMapIoSpace` является функцией менеджера памяти режима ядра, которая отображает диапазон физического адреса на диапазон адресов в виртуальной нестраничной памяти. Драйвер использует возвращенные виртуальные адреса для доступа к отображенными ресурсам.

Дополнительную информацию по этому вопросу см. в разделе **MmMapIoSpace** на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81589>. Также информацию о макросах `PORT` и `REGISTER` см. в разделе **Hardware Abstraction Layer Routines** (Процедуры уровня абстрагирования от оборудования) на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=81591>.

Для ресурсов прерываний объект прерываний WDF сам подбирает свои ресурсы, без участия драйвера. Драйвер создает объект прерывания в своей функции обратного вызова `EvtDriverDeviceAdd`, а инфраструктура управляет процессом выделения ресурсов для прерывания и подключения прерывания. Информацию о создании объекта прерывания и обслуживания прерываний см. в разд. "Прерывания и их обработка" далее в этой главе.

## Пример: отображение ресурсов

Устройство PCI, поддерживаемое образцом драйвера `Pcidrv`, имеет три регистра BAR (Base Address Register, регистр базового адреса):

- ◆ регистр BAR 0 отображается на пространство памяти;
- ◆ регистр BAR 1 отображается на пространство ввода/вывода;
- ◆ регистр BAR 3 отображается на пространство флэш-памяти.

Драйвер определяет, какой регистр BAR — отображенный на пространство ввода/вывода или пространство памяти — использовать для доступа к регистрам управления и статуса. Образец драйвера проверяет на наличие регистров как в адресном пространстве памяти, так и в адресном пространстве ввода/вывода. На некоторых платформах регистры ввода/вывода могут быть отображены на пространство памяти. Все драйверы должны поддерживать работу с такими регистрами.

В образце драйвера код для отображения ресурсов изолирован в функции `NICMapHwResources`, которая вызывается в функции обратного вызова `EvtDevicePrepareHardware` драйвера. Функции `NICMapHwResources` передается два параметра: указатель на область контекста устройства (т. е. `FdoData`) и дескриптор списка преобразованных ресурсов (т. е. `ResourcesTranslated`), который был передан функции `EvtDevicePrepareHardware`. Драйвер использует преобразованные ресурсы для отображения регистров устройства на пространство ввода/вывода и пространство памяти.

Исходный код этого примера (листинг 16.1) взят из файла `Pcidrv\Sys\Hw\Nic_init.c`.

### Листинг 16.1. Отображение аппаратных ресурсов

```

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; i<WdfCmResourceListGetCount(ResourcesTranslated);i++) {
        descriptor = WdfCmResourceListGetDescriptor(ResourcesTranslated, i);
        if (!descriptor) {
            return STATUS_DEVICE_CONFIGURATION_ERROR;
        }
        switch (descriptor->Type) {
            case CmResourceTypePort:
                numberofBARs++;
                . . . // Код опущен для краткости.
                // На этой системе порт находится в пространстве ввода/вывода.
                FdoData->IoBaseAddress =
                    ULongToPtr(descriptor->u. Port. Start. Low/Part);
                FdoData->IoRange = descriptor->u.Port.Length;
                FdoData->ReadPort = NICReadPortUShort;
                FdoData->WritePort = NICWritePortUShort;
                bResPort = TRUE;
                FdoData->MappedPorts = FALSE;
                break;
            case CmResourceTypeMemory:
                numberofBARs++;
                if (numberOfBARs == 1) {
                    // Размер пространства памяти CSR должен быть 0x1000.
                    ASSERT(descriptor->u.Memory.Length == 0x1000);
                    FdoData->MemPhysAddress = descriptor->u.Memory.Start;
                    FdoData->CSRAddress = MmMapIoSpace(
                        descriptor->u.Memory.Start,
                        NIC_MAP_IOSPACE_LENGTH,
                        MmNonCached);
                    bResMemory = TRUE;
                }
                else if (numberOfBARs == 2){
                    // Отражаем физические адреса на виртуальные и
                    // используем макросы семейства READ/WRITE_REGISTER_xxx.
                    FdoData->IoBaseAddress = MmMapIoSpace(
                        descriptor->u.Memory.Start,
                        descriptor->u.Memory.Length,
                        MmNonCached);
                    FdoData->ReadPort = NICReadRegisterUShort;
                    FdoData->WritePort = NICWriteRegisterUShort;
                    FdoData->MappedPorts = TRUE;
                    bResPort = TRUE;
                }
        }
    }
}

```

```
else if (numberOfBARs == 3) {
    // Доступа к флэш-памяти нет.
    . . . // Код опущен для краткости.
}
else {
    status = STATUS_DEVICE_CONFIGURATION_ERROR;
    return status;
}
break;
case CmResourceTypeInterrupt:
    . . . // Код опущен для краткости.
default:
    . . . // Код опущен для краткости.
}
}

// Удостоверяемся в том, что имеются в наличии все ресурсы.
if (!(bResPort && bResInterrupt && bResMemory)) {
    status = STATUS_DEVICE_CONFIGURATION_ERROR;
}
. . . // Код опущен для краткости.
return status;
}
```

Драйвер шины использует сырье ресурсы для организации работы устройства, но функциональный драйвер обычно использует только преобразованные ресурсы. Функциональный драйвер в приведенном листинге анализирует список преобразованных ресурсов в цикле, который начинается с нуля и заканчивается по достижению последнего ресурса в списке. Драйвер определяет число ресурсов в списке, вызывая функцию `WdfCmResourceListGetCount`.

Для каждого ресурса в списке драйвер вызывает метод `WdfCmResourceListGetDescriptor`, чтобы получить указатель на структуру дескриптора ресурса.

Для ресурсов типа `CmResourceTypePort` драйвер сохраняет начальный адрес и диапазон адресов в области контекста устройства, после чего задает адреса функций, которые он использует для обращения к ресурсам порта. Функции `NICReadPortUShort` и `NICWritePortUShort` являются оберточными интерфейсами для макросов `READ_PORT USHORT` и `WRITE_PORT USHORT` в уровне HAL.

Для ресурсов типа `CmResourceTypeMemory` драйвер также сохраняет начальный адрес и диапазон адресов в области контекста устройства, но использует процедуру `MmMapIoSpace` для отображения ресурсов и получения виртуального адреса для доступа к ним. Процедура `MmMapIoSpace` возвращает указатель на базовый виртуальный адрес отображеного диапазона, который драйвер сохраняет вместе со значением размера диапазона в области контекста устройства. Драйвер также сохраняет указатели на функции, которые он использует для чтения и записи в ресурсы. Для ресурсов, отображенных на пространство памяти, драйвер организовывает функции `NICReadRegisterUShort` и `NICWriteRegisterUShort`, которые являются оберточными интерфейсами для макросов `READ_REGISTER USHORT` и `WRITE_REGISTER USHORT` уровня HAL.

Для ресурсов типа `CmResourceTypeInterrupt` драйверу не требуется сохранять информацию о ресурсах, т. к. инфраструктура делает это прозрачно для драйвера. Образец драйвера просто проверяет этот ресурс на полноту.

## Пример: отмена отображения ресурсов

При удалении устройства или при его остановке для перераспределения системных ресурсов, драйвер должен освободить свои отображеные ресурсы в функции обратного вызова `EvtDeviceReleaseHardware`. Инфраструктура вызывает эту функцию после вызова функции `EvtDeviceD0Exit` драйвера.

Образец драйвера `Pcidrv` отменяет отображение аппаратных ресурсов во внутренней функции `NICUnmapHwResources`, которая вызывается в функции обратного вызова `EvtDeviceReleaseHardware` драйвера. Сама функция `NICUnmapHwResources` определена в исходном файле `Pcidrv\sys\hw\nic_init.c`. Код для отмены отображения ресурсов показан в листинге 16.2.

### Листинг 16.2. Отмена отображения аппаратных ресурсов

```
if (FdoData->CSRAddress) {
    MmUnmapIoSpace(FdoData->CSRAddress, NIC_MAP_IOSPACE_LENGTH);
    FdoData->CSRAddress = NULL;
}
if (FdoData->MappedPorts) {
    MmUnmapIoSpace(FdoData->IoBaseAddress, FdoData->IoRange);
    FdoData->IoBaseAddress = NULL;
}
```

Если образец драйвера имеет отображеные ранее ресурсы, поле `CSRAddress` области контекста устройства содержит действительный указатель, а поле `MappedPorts` имеет значение `TRUE`. Драйвер вызывает процедуру `MmUnmapIoSpace`, чтобы отменить отображение диапазона адресов с адреса `CSRAddress` и диапазона адресов с адреса `IoBaseAddress`. После этого драйвер устанавливает значения соответствующих полей области контекста устройства равными `NULL`.

## Прерывания и их обработка

Некоторые устройства, подключаемые к шине PCI, шине PCI Express или другой монтажнойшине, генерируют прерывания, чтобы дать знать процессору — и, таким образом, операционной системе — что они нуждаются в обслуживании. Функциональный драйвер для таких устройств должен содержать код для разрешения и запрещения прерывания в аппаратном обеспечении устройства, и для реагирования на прерывания, возникающие в процессе работы устройства (обычно сигнализирующие, что устройство завершило требуемую операцию).

Устройства, подключаемые к протокольным шинам, таким как USB и IEEE 1394, и по Bluetooth, не генерируют прерываний, поэтому для их драйверов не требуется код для обработки прерываний.

### ◆ Прерывания на основе линий запроса и на основе сообщений.

Большинство устройств выдают прерывания, посылая электрический сигнал по выделенной линии, называемой линией прерывания. Некоторые более новые устройства PCI вместо выдачи прерываний по линии запроса генерируют прерывания MSI (Message Signaled Interrupt, прерывание, инициируемое сообщением), записывая данные по определенному адресу. Версии Windows до Windows Vista поддерживают только прерывания на основе линий запроса. Windows Vista и более поздние версии Windows поддерживают

как прерывания с использованием линий запроса, так и прерывания, инициируемые сообщениями.

### Полезная информация

Компания Microsoft внесла несколько улучшений в структуру прерываний Windows Vista, с которыми можно ознакомиться в статье "Interrupt Architecture Enhancements in Windows Vista" ("Улучшения в архитектуре прерываний в Windows Vista") по адресу <http://go.microsoft.com/fwlink/?LinkId=81584>.

#### ◆ Драйверная поддержка для обработки прерываний.

Независимо от типа прерываний, генерируемых устройством, для их обработки функциональным драйверам требуются одинаковые объекты и функции обратного вызова. Чтобы реализовать поддержку обработки в драйвере, необходимо выполнить следующие действия.

- Создать объект прерывания для всех прерываний на основе линии запроса или на основе сообщений, которые может выдавать устройство.
- Необязательно, предоставить функции обратного вызова *EvtInterruptEnable* и *EvtInterruptDisable*, которые разрешают и запрещают прерывания в аппаратной части устройства соответственно.
- Дополнительно, если во время переходов между состояниями энергопотребления при разрешенных прерываниях устройству требуется выполнить какие-либо операции, предоставить функции обратного вызова *EvtDeviceD0EntryPostInterruptsEnabled* и *EvtDeviceD0ExitPreInterruptsDisabled*.
- Предоставить функцию обратного вызова *EvtInterruptIsr* для обработки прерываний на уровне DIRQL.
- Дополнительно, если драйверу требуются выполнить дополнительные задачи обработки прерываний на уровне IRQL = DISPATCH\_LEVEL, предоставить функцию обратного вызова *EvtInterruptDpc*.

Функции обратного вызова *EvtInterruptIsr*, *EvtInterruptEnable* и *EvtInterruptDisable* исполняются на уровне DIRQL. Во время исполнения этих функций прерывания на уровне IRQL  $\leq$  DIRQL маскируются и, таким образом, не могут происходить. Поэтому эти функции должны выполнять только абсолютно необходимую обработку. Выполнение длительных операций в этих функциях обратного вызова может замедлить систему.

## Объекты прерывания

Объект прерывания (т. е. объект типа *WDFINTERRUPT*) представляет вектор прерывания или отдельное прерывание MSI. Объекты прерывания создаются драйвером во время исполнения функции обратного вызова *EvtDriverDeviceAdd*. Каждый объект прерывания должен содержать указатели на функции обратного вызова *EvtInterruptIsr* и *EvtInterruptDpc*, а также может содержать дополнительную информацию. Для создания объекта прерывания драйвер заполняет структуру атрибутов и структуру конфигурации, после чего вызывает метод для создания объекта.

Если при прерывании устройство генерирует временные данные, получаемые функцией обратного вызова *EvtInterruptIsr* и используемые функцией *EvtInterruptDpc*, для объекта пре-

рывания необходимо инициализировать структуру атрибутов объекта и создать область контекста объекта. Функция обратного вызова может получить данные на уровне DIRQL и сохранить их в области контекста объекта, где к ним может обращаться функция *EvtInterruptDpc* и другие функции семейства *EvtInterruptXxx*.

## Структура конфигурации объекта прерывания

Для инициализации структуры *WDF\_INTERRUPT\_CONFIG* указателями на функции обратного вызова *EvtInterruptIsr* и *EvtInterruptDpc*, которые драйвер реализует для прерывания, драйвер вызывает функцию *WDF\_INTERRUPT\_CONFIG\_INIT*. Прежде чем создать объект прерывания, драйвер также может сохранить в этой структуре дополнительную информацию. Наиболее широко применяются следующие поля структуры:

- ◆ *SpinLock* — необязательный дескриптор объекта спин-блокировки WDF;
- ◆ *AutomaticSerialization* — булево значение для разрешения и запрещения сериализации функции обратного вызова *EvtInterruptDpc*;
- ◆ *EvtInterruptEnable* и *EvtInterruptDisable* — указатели на функции обратного вызова драйвера для разрешения и запрещения прерываний в аппаратном обеспечении устройства.

### Поле *SpinLock*

Если драйвер создает несколько объектов прерываний и должен синхронизировать свою обработку нескольких прерываний, как может быть необходимым для поддержки прерываний MSI, для всех этих объектов прерываний можно использовать одну предоставляемую драйвером спин-блокировку. Драйвер создает блокировку, вызывая метод *WdfSpinLockCreate*, после чего предоставляет дескриптор объекта спин-блокировки в поле *SpinLock* структуры конфигурации прерывания для каждого объекта прерывания, который будет защищен данной блокировкой. Инфраструктура определяет наивысший уровень DIRQL среди всех прерываний и передает значение этого DIRQL, когда она вызывает систему для подключения прерывания. Система всегда захватывает блокировку на этом уровне DIRQL, чтобы никакие ассоциированные прерывания могли прервать исполнение синхронизированного кода.

### Поле *AutomaticSerialization*

Поле *AutomaticSerialization* указывает, сериализует ли инфраструктура вызовы функций обратного вызова *EvtInterruptDpc* с вызовами функций обратного вызова для событий ввода/вывода для объекта устройства и с функциями обратного вызова для всех других объектов, для которых драйвер установил автоматическую сериализацию. Если драйвер должен всегда сериализовывать исполнение своей функции обратного вызова *EvtInterruptDpc* с исполнением другой функции *EvtDpcFunc* или с исполнением любых других функций обратного вызова для события ввода/вывода, к которым применима инфраструктурная синхронизация, то необходимо разрешить автоматическую сериализацию. Если такая синхронизация требуется драйверу только изредка, то следует использовать спин-блокировку.

## Атрибуты объекта прерывания

Кроме инициализации структуры конфигурации объекта прерывания, драйвер также инициализирует структуру атрибутов объекта. Некоторые драйверы могут пропустить этот шаг и просто передать константу *WDF\_NO\_ATTRIBUTES*. Но если драйверу требуются данные контекста прерывания для нескольких его функций обратного вызова семейства *EvtInterruptXxx*,

то он должен создать область контекста и инициализировать поле `ContextTypeInfo` в структуре атрибутов, вызывая для этого метод `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE`.

Например, образец драйвера AMCC5933 определяет тип `INTERRUPT_DATA` для области контекста прерывания и инициализирует структуру атрибутов объекта этой информацией следующим образом:

```
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&interruptAttributes,  
    INTERRUPT_DATA);
```

Поля `SynchronizationScope` и `ExecutionLevel` структуры атрибутов неприменимы к объектам прерываний. Драйвер не должен устанавливать поле `Parent`, т. к. инфраструктура устанавливает объект устройства в качестве родителя для всех объектов прерываний и драйвер не может изменить эту установку.

### Создание объекта прерывания

Драйвер создает объект прерывания, вызывая метод `WdfInterruptCreate`. В качестве параметров методу передаются дескриптор объекта устройства, указатель на структуру конфигурации прерывания и указатель на структуру атрибутов. Метод возвращает значение статуса и дескриптор созданного объекта прерывания.

Если устройство может выдавать несколько векторов или сообщений прерываний, то драйвер должен сконфигурировать и создать объект прерывания для каждого из них. Менеджер PnP пытается назначить все векторы прерываний или сообщения прерываний, которые устройство может поддерживать. В случае с сообщениями MIS, если менеджер PnP не может назначить все сообщения, то он назначает только одно сообщение. Инфраструктура не использует ни одного из оставшихся объектов прерываний и не вызывает их функций обратного вызова.

Драйвер KMDF не обязан подключать и отключать прерывания. Инфраструктура автоматически подключает и отключает прерывания для драйвера, как часть процесса перехода в состояние D0 и выхода из него. Драйверам требуется только создать объекты прерываний и предоставить функции обратного вызова для разрешения, запрещения и обслуживания прерываний.

Образец драйвера `Pcidrv` создает объект прерывания в функции `NICAllocateSoftwareResources`, которая вызывается в функции обратного вызова `EvtDriverDeviceAdd` драйвера. В листинге 16.3 показан пример создания драйвером `Pcidrv` объекта прерывания. Этот исходный код находится в заголовочном файле `Pcidrv\sys\hw\nic_init.c`.

#### Листинг 16.3. Создание объекта прерывания

```
WDF_INTERRUPT_CONFIG_INIT(&interruptConfig,  
    NICEvtInterruptIsr,  
    NICEvtInterruptDpc);  
interruptConfig.EvtInterruptEnable = NICEvtInterruptEnable;  
interruptConfig.EvtInterruptDisable = NICEvtInterruptDisable;  
status = WdfInterruptCreate(FdoData->WdfDevice,  
    &interruptConfig,  
    WDF_NO_OBJECT_ATTRIBUTES,  
    &FdoData->WdfInterrupt);  
if (!NT_SUCCESS (status)) {  
    return status;  
}
```

Драйвер Pcidrv инициализирует структуру конфигурации прерывания, задавая указатели на функции обратного вызова для событий *EvtInterruptIsr* и *EvtInterruptDpc*, называющиеся *NICEvtInterruptIsr* и *NICEvtInterruptDpc* соответственно. Драйвер также устанавливает указатели на функции обратного вызова для событий *EvtInterruptEnable* и *EvtInterruptDisable*, называющиеся *NICEvtInterruptEnable* и *NICEvtInterruptDisable* соответственно. После этого драйвер создает объект прерывания, вызывая метод *WdfInterruptCreate*. В качестве параметров методу передаются структура конфигурации прерывания, константа *WDF\_NO\_ATTRIBUTES* и указатель на переменную для получения дескриптора объекта прерывания. Драйвер сохраняет этот дескриптор в области контекста объекта своего устройства.

## Разрешение и запрещение прерываний

Если устройство может генерировать прерывания, то драйвер должен обладать способностью разрешать и запрещать прерывания в аппаратном обеспечении. Для разрешения и запрещения прерываний большинство драйверов должны реализовывать функции обратного вызова *EvtInterruptEnable* и *EvtInterruptDisable* соответственно. Инфраструктура активирует эти функции на уровне DIRQL при захваченной спин-блокировке для прерывания и вызывает каждую из этих функций по одному разу для каждого прерывания, которое может генерировать устройство. Применение этих функций является наиболее безопасным подходом и сводит к минимуму возможность ошибок.

Но разрешение и запрещение каждого прерывания по отдельности является неудобным для некоторых драйверов по причине особенностей конструкции их устройств. Для таких устройств эти задачи следует выполнять в функциях обратного вызова *EvtDeviceD0EntryPostInterruptsEnabled* и *EvtDeviceD0ExitPreInterruptsDisabled*. Эти функции исполняются на уровне PASSIVE\_LEVEL, поэтому драйвер должен обезопасить свой доступ к аппаратному обеспечению устройства посредством захвата спин-блокировки для прерывания и выполняя любую другую синхронизацию, требуемую конструкцией устройства.

Инфраструктура вызывает функцию обратного вызова *EvtInterruptEnable* при следующих обстоятельствах:

- ◆ при переходе устройства в состояние D0 после возвращения управления функцией *EvtDeviceD0Entry*;
- ◆ в ответ на вызов драйвером метода *WdfInterruptEnable*.

Функция обратного вызова *EvtInterruptEnable* исполняется на уровне DIRQL, поэтому она должна лишь быстро разрешить прерывание и возвратить управление. Если после разрешения прерывания драйверу требуется выполнить дополнительную обработку, он должен зарегистрировать функцию обратного вызова *EvtDeviceD0EntryPostInterruptsEnabled*, которую инфраструктура вызывает на уровне IRQL = PASSIVE\_LEVEL.

Функция обратного вызова *EvtInterruptEnable* вызывается с двумя параметрами: дескриптором объекта прерывания и дескриптором ассоциированного объекта устройства. Если драйвер сохранил информацию о регистрах своего устройства в области контекста объекта устройства, он может использовать дескриптор объекта устройства для вызова функции доступа к области контекста. Используя возвращенный указатель на область контекста, драйвер может получить доступ к аппаратным регистрам, требуемый для разрешения прерывания.

Функция обратного вызова *EvtInterruptDisable* драйвера запрещает прерывания для его устройства. Инфраструктура вызывает эту функцию при каждом вызове драйвером метода *WdfInterruptDisable* и при выходе устройства из состояния D0, но перед вызовом функции

*EvtDeviceD0Exit*. Функция *EvtInterruptDisable* вызывается на уровне DIRQL устройства и с захваченной спин-блокировкой прерывания; поэтому она должна ограничить свои действия запрещением прерывания, после чего возвратить управления. Если перед запрещением прерывания драйверу требуется выполнить дополнительную обработку, он должен зарегистрировать функцию обратного вызова *EvtDeviceD0ExitPreInterruptsDisabled*, которую инфраструктура вызывает на уровне PASSIVE\_LEVEL.

Функции *EvtInterruptDisable* передаются те же два параметра, что и для функции обратного вызова *EvtInterruptEnable*, с помощью которых она отменяет действия, выполненные в первой функции.

В листинге 16.4 приведен слегка модифицированный код из файла Isrdpc.c для разрешения прерываний в образце драйвера Pcidrv. Драйвер устанавливает регистры устройства, вызывая макрос *NICEnableInterrupt*. Вместо вызова этого макроса, в листинге 16.4 приводятся релевантные операторы самого макроса.

#### Листинг 16.4. Разрешение прерываний

```
NTSTATUS NICEvtInterruptEnable(
    IN WDFINTERRUPT Interrupt,
    IN WDFDEVICE     AssociatedDevice)
{
    PFDO_DATA fdoData;
    fdoData = FdoGetData(AssociatedDevice);
    fdoData->CSRAddress->ScbCommandHigh = 0;
    return STATUS_SUCCESS;
}
```

С помощью дескриптора объекта устройства драйвер получает указатель на область контекста объекта устройства, где он сохранил информацию о регистрах устройства. После этого драйвер разрешает прерывания, устанавливая регистр устройства PCI.

В листинге 16.5 приводится исходный код функции обратного вызова для события *EvtInterruptDisable*, который находится в том же самом файле. Как можно видеть, он похож на код функции для разрешения прерываний. Так, вместо вызова макроса *NICDisableInterrupt* в листинге 16.4 приведены релевантные операторы самого макроса.

#### Листинг 16.5. Запрещение прерываний

```
NTSTATUS NICEvtInterruptDisable(
    IN WDFINTERRUPT Interrupt,
    IN WDFDEVICE     AssociatedDevice)
{
    PFDO_DATA fdoData;
    fdoData = FdoGetData(AssociatedDevice);
    fdoData->CSRAddress->ScbCommandHigh = SCB_INT_MASK;
    return STATUS_SUCCESS;
}
```

В образце драйвера Pcidrv, чтобы запретить прерывания, функция обратного вызова *EvtInterruptDisable* устанавливает регистр устройства PCI.

## Обработка после разрешения и до запрещения прерываний

Некоторые устройства вызывают шторм прерываний, если они инициализируются после того, как были разрешены их прерывания. Поэтому драйвер такого устройства должен быть в состоянии выполнить инициализацию после перехода устройства в состояние D0, но перед тем, как выполнено разрешение его прерывания. Но также есть устройства, для которых нельзя выполнить инициализацию до тех пор, пока не будет разрешено прерывание. Чтобы позволить устройствам обоих типов работать должным образом, инфраструктура предоставляет события после разрешения прерывания и до запрещения прерываний, для которых драйвер может зарегистрировать функции обратного вызова.

При включении питания устройства инфраструктура активирует обратные вызовы драйвера в следующем порядке:

1. *EvtDeviceD0Entry*.
2. *EvtInterruptEnable*.
3. *EvtDeviceD0EntryPostInterruptsEnabled*.

Сначала инфраструктура вызывает функцию *EvtDeviceD0Entry* на уровне IRQL = PASSIVE\_LEVEL. Драйверы, которые должны инициализировать свои устройства до подключения прерывания, например, чтобы предотвратить шторм прерываний, должны выполнить эту инициализацию в этом обратном вызове. Следующей инфраструктура вызывает функцию *EvtInterruptEnable*. В этой функции драйверы должны только разрешить свои прерывания и не выполнять никаких, или минимальное число, других действий, т. к. эта функция вызывается на уровне IRQL = DIRQL. Наконец, инфраструктура вызывает функцию *EvtDeviceD0EntryPostInterruptsEnabled* на уровне IRQL = PASSIVE\_LEVEL. Драйверы, которые должны инициализировать свои устройства после подключения прерывания, должны выполнить эту инициализацию в этом обратном вызове.

При отключении питания устройства инфраструктура вызывает соответствующие функции в обратном порядке:

1. *EvtDeviceD0ExitPreInterruptsDisabled*.
2. *EvtInterruptDisable*.
3. *EvtDeviceD0Exit*.

Для отмены действий, выполненных в функции обратного вызова *EvtDeviceD0EntryPostInterruptsEnabled*, драйвер регистрирует функцию обратного вызова *EvtDeviceD0ExitPreInterruptsDisabled*. Подобно функции, выполняющейся после разрешения прерывания, функция, выполняющаяся до запрещения прерывания, исполняется на уровне PASSIVE\_LEVEL, подготавливая запрещение прерывания.

Функции обратного вызова для выполнения обработки после разрешения и до запрещения прерывания драйвер регистрирует в структуре *WDF\_PNPPOWER\_EVENT\_CALLBACKS* перед созданием объекта устройства. Образец драйвера заполняет эту структуру в функции обратного вызова *EvtDriverDeviceAdd* следующим образом:

```
pnpPowerCallbacks.EvtDeviceD0EntryPostInterruptsEnabled =
    NICEvtDeviceD0EntryPostInterruptsEnabled;
pnpPowerCallbacks.EvtDeviceD0ExitPreInterruptsDisabled =
    NICEvtDeviceD0ExitPreInterruptsDisabled;
```

В текущей версии образца драйвера обе функции реализованы лишь в виде заполнителей.

## Процедуры обслуживания прерываний

Когда устройство выдает прерывание, Windows вызывает драйвер, чтобы обслужить это прерывание. Но один вектор прерывания или одно сообщение MSI может использоваться несколькими устройствами. Windows содержит внутренний список процедур ISR (Interrupt Service Routine, процедура обслуживания прерывания) для устройств, выдающих прерывания на одинаковом уровне. Когда прибывает сигнал прерывания, Windows проходится по этому списку, последовательно вызывая указанные в нем драйверы, пока один из них не признает и не обслужит прерывание.

Инфраструктура перехватывает вызов от операционной системы и вызывает функцию обратного вызова *EvtInterruptIsr*, зарегистрированную драйвером. Спин-блокировка прерывания предотвращает другие прерывания на уровне IRQL ≤ DIRQL, чтобы функция обратного вызова *EvtInterruptIsr* могла получить из устройства временные, специфичные для прерывания, данные.

Функция обратного вызова *EvtInterruptIsr* исполняется на уровне DIRQL, поэтому она должна выполнять только следующие операции и никаких больше:

- ◆ определить, выдает ли устройство прерывание, и если не выдает, то немедленно возвратить значение FALSE;
- ◆ остановить подачу прерывания устройством;
- ◆ скопировать все временные данные из устройства в разделяемое хранилище, обычно в область контекста объекта прерывания;
- ◆ поставить в очередь процедуру DPC для выполнения работы, связанной с прерыванием.

Функция *EvtInterruptIsr* вызывается с дескриптором объекта прерывания для устройства драйвера и значением *ULONG*. Если устройство сконфигурировано для прерываний MSI, то это значение указывает идентификатор сообщения; в противном случае оно равно нулю.

Функция *EvtInterruptIsr* исполняется на том же самом процессоре, на который ее устройство подало прерывание; в свою очередь, ее функция *EvtInterruptDpc* или *EvtDpcFunc* исполняется на том же самом процессоре, что и функция *EvtInterruptIsr*, которая поставила ее в очередь.

Чтобы определить, является ли его устройство источником прерывания, драйвер должен обращаться к аппаратному обеспечению устройства. Поэтому драйвер должен был перед этим сохранить указатель на аппаратные регистры в области, к которой он может иметь доступ из функции *EvtInterruptIsr* на уровне DIRQL. Область контекста объекта прерывания является хорошим выбором в качестве такой области хранения, т. к. функция обратного вызова *EvtInterruptIsr* получает дескриптор объекта прерывания от инфраструктуры. Драйвер также может использовать для этой цели область контекста объекта устройства. К объекту устройства можно обращаться на уровне DIRQL, вызывая метод *WdfInterruptGetDevice*, поэтому образец драйвера PCIDRV сохраняет указатель в области контекста объекта устройства.

После получения драйвером указателя на аппаратные регистры, он проверяет аппаратное обеспечение устройства, чтобы выяснить, не является ли его устройство источником прерывания. Если прерывание не было инициировано устройством драйвера, драйвер немедленно возвращает значение FALSE. Функция *EvtInterruptIsr* не должна возвращать FALSE, если прерывание было выдано ее устройством. Такие "непризнанные" прерывания могут со временем вызвать зависание или сбой системы.

Если прерывание было выдано его устройством, то драйвер останавливает подачу прерывания устройством и копирует все временные данные в область контекста объекта прерывания или в какое-либо другое место, доступное для совместного использования его функциями *EvtInterruptIsr* и *EvtInterruptDpc*. После этого он вызывает метод *WdfInterruptQueueDpcForIsr*, чтобы поставить в очередь процедуру DPC, и возвращает значение TRUE.

### Внимание!

Функция обратного вызова драйвера для обслуживания прерывания должна быть написана таким образом, чтобы она могла пользоваться векторами прерываний или прерываниями MSI совместно с другими устройствами. Функция не должна предполагать, что она вызывается только для прерываний, выдаваемых ее устройством. Но она должна возвращать TRUE, только если прерывание подается ее устройством. Возвращение TRUE при любых других условиях может вызвать зависание системы.

В листинге 16.6 приведен исходный код определения функции обратного вызова *EvtInterruptIsr* в файле *Pcidrv\sys\hw\isrdpc.c* для образца драйвера *Pcidrv*.

#### Листинг 16.6. Функция обратного вызова *EvtInterruptIsr* для обслуживания прерывания

```
BOOLEAN NICEvtInterruptIsr(
    IN WDFINTERRUPT Interrupt,
    IN ULONG MessageID)
{
    BOOLEAN InterruptRecognized = FALSE;
    PFDO_DATA FdoData = NULL;
    USHORT IntStatus;
    UNREFERENCED_PARAMETER(MessageID);
    FdoData = FdoGetData(WdfInterruptGetDevice(Interrupt));
    // Если прерывание разрешено и активировано, обработать прерывание.
    if (!NIC_INTERRUPT_DISABLED(FdoData) &&
        NIC_INTERRUPT_ACTIVE(FdoData)) {
        InterruptRecognized = TRUE;
        // Запретить прерывание.
        // Оно будет снова разрешено в функции NICEvtInterruptDpc.
        NICDisableInterrupt(FdoData);
        // Подтвердить прерывание/прерывания и получить значение статуса.
        NIC_ACK_INTERRUPT(FdoData, IntStatus);
        WdfInterruptQueueDpcForIsr(Interrupt);
    }
    return InterruptRecognized;
}
```

Первый шаг драйвера *Pcidrv* в том, чтобы определить, является ли его устройство источником текущего прерывания. Для этого драйвер должен выполнить проверку регистров своего устройства. Драйвер получает дескриптор объекта устройства, ассоциированного с объектом прерывания, вызывая метод *WdfInterruptGetDevice*. Полученный дескриптор потом передается функции *FdoGetData*, чтобы получить указатель на область контекста устройства, в которой драйвер ранее сохранил указатель на свои отображенные аппаратные регистры.

Для большинства драйверов проверка, запрещены ли прерывания устройства, не является обязательной. Но в данном случае драйвер определяет макросы *NIC\_INTERRUPT\_DISABLED* и *NIC\_INTERRUPT\_ACTIVE* в заголовочном файле *Nic\_def.h*. Эти макросы проверяют аппаратные

регистры, чтобы определить, были ли разрешены прерывания для устройства и активны ли они в настоящее время. Если прерывания запрещены, то устройство драйвера не может быть источником текущего прерывания. То же самое относится и к случаю, когда прерывание устройства разрешено, но в данный момент не активно. В любом из этих случаев драйвер возвращает значение FALSE в переменной `InterruptRecognized`.

Но если прерывания разрешены и в данный момент прерывание активно, то устройство должно быть источником текущего прерывания. В этом случае драйвер устанавливает значение переменной `InterruptRecognized` в TRUE.

Чтобы остановить подачу прерывания устройством, драйвер вызывает функцию `NICDisableInterrupt`, после чего с помощью определенного в драйвере макроса `NIC_ACK_INTERRUPT` подтверждает прерывание в аппаратном обеспечении. Наконец, драйвер ставит в очередь на исполнение процедуру обратного вызова `EvtInterruptDpc`, вызывая метод `WdfInterruptQueueDpcForIsr`, после чего возвращает управление.

## Отложенная обработка прерываний

Когда исполняется процедура DPC, инфраструктура вызывает функцию обратного вызова `EvtInterruptDpc` драйвера. Эта функция выполняет специфичную для устройства обработку прерывания и снова разрешает прерывание устройства. Функция обратного вызова исполняется на уровне IRQL = DISPATCH\_LEVEL и поэтому не должна ни пытаться выполнять никакие операции, которые могли бы вызвать страничную ошибку, ни ожидать никаких объектов диспетчера.

В листинге 16.7 приведен исходный код определения функции обратного вызова `EvtInterruptDpc` в файле `Isrdpc.c` для образца драйвера `Pcidrv`.

### Листинг 16.7. Отложенная обработка прерывания в функции обратного вызова `EvtInterruptIsr`

```
VOID NICEvtInterruptDpc(
    IN WDFINTERRUPT WdfInterrupt,
    IN WDOBJECT     WdfDevice)
{
    PFDO_DATA fdoData = NULL;
    fdoData = FdoCetData(WdfDevice);
    . . . // Код, специфичный для устройства, опущен.
    // Снова разрешаем прерывание. Этот драйвер был перенесен с WDM,
    // поэтому он использует WdfInterruptSynchronize. Вместо этого,
    // он мог бы захватить спин-блокировку прерывания, вызвав метод
    // WdfInterruptAcquireLock.
    WdfInterruptSynchronize (WdfInterrupt,
                            NICEnableInterrupt,
                            fdoData);
    return;
}
```

Функция обратного вызова `EvtInterruptDpc` драйвера `Pcidrv` обрабатывает результаты операции ввода/вывода, после чего снова разрешает прерывание. Драйвер должен отменить запрещение прерывания на уровне DIRQL, удерживая при этом спин-блокировку прерывания.

Так как этот образец драйвера был перенесен с WDM, то в нем применяется метод `WdfInterruptSynchronize`. Инфраструктура поддерживает этот метод главным образом для совместимости с существующими драйверами WDM. В качестве параметров метод `WdfInterruptSynchronize` принимает дескриптор объекта прерывания, указатель на функцию для события `EvtInterruptSynchronize`, которая должна исполняться на уровне DIRQL (т. е. функция `NICEnableInterrupt`), и указатель на данные контекста, определенные драйвером. Метод `WdfInterruptSynchronize` сначала вызывает систему, чтобы получить spin-блокировку прерывания, после чего вызывает функцию `EvtInterruptSynchronize`, передавая ей указатель на контекст. Когда функция `EvtInterruptSynchronize` завершает свою работу, система освобождает spin-блокировку и метод `WdfInterruptSynchronize` возвращает управление.

Лучшим способом синхронизации обработки на уровне DIRQL было бы получить спин-блокировку для прерывания напрямую, как описывается в следующем разделе.

## Синхронизация обработки на уровне DIRQL

Для синхронизации обработки на уровне DIRQL драйвер может вызвать метод `WdfInterruptAcquireLock`, чтобы захватить спин-блокировку прерывания, непосредственно перед тем моментом, когда коду требуется синхронизация, а сразу же после синхронизированного кода вызвать метод `WdfInterruptReleaseLock`. Метод `WdfInterruptAcquireLock` повышает уровень IRQL на текущем процессоре до уровня DIRQL и захватывает спин-блокировку прерывания. А метод `WdfInterruptReleaseLock` освобождает блокировку и понижает уровень IRQL.

В листинге 16.8 показано, каким образом драйвер PLX9x5x применяет эту блокировку. Код примера взят из файла `SysMsrDpc.c`.

### Листинг 16.8. Использование spin-блокировки прерывания

```
WdfInterruptAcquireLock(Interrupt);
if ((devExt->IntCsr.bits.DmaChan0IntActive)
    && (devExt->Dma0Csr.bits.Done)) {
    // Если прерывание выдается каналом 0 (запись) Dma0
    // и в Dma0 CSR установлен бит Done, то завершена операция WRITE.
    // Очищаем бит done и бит прерывания канала.
    devExt->IntCsr.bits.DmaChan0IntActive = FALSE;
    devExt->Dma0Csr.uchar = 0;
    writeInterrupt = TRUE;
}
. . . // Код, специфичный для устройства, опущен.
WdfInterruptReleaseLock( Interrupt );
```

В листинге 16.8 приведен исходный код из функции обратного вызова `EvtInterruptDpc` об разца драйвера `Pcidrv`. Устройство PLX имеет два канала DMA: один для чтения, а второй для записи. Каналы могут обрабатывать транзакции DMA одновременно. При завершении транзакции DMA устройство выдает прерывание. Функция `EvtInterruptIsr` драйвера определяет канал, выдавший прерывание, запрещает дальнейшие прерывания на этом канале и ставит в очередь функцию обратного вызова `EvtInterruptDpc`. Это делает возможным исполнение процедуры DPC для одного канала, в то время как процедура ISR обслуживает прерывание на другом канале. Для синхронизации доступа к аппаратным регистрам совместно с процедурой ISR процедура DPC захватывает прерывание. Процедура DPC проверяет аппаратные регистры устройства, чтобы определить причину, по которой устройство выдало прерывание, после чего очищает эти регистры, чтобы отменить запрет прерываний.

# ГЛАВА 17

## Прямой доступ к памяти

Применение прямого доступа к памяти (Direct Memory Access, DMA) для обмена данными с устройством предоставляет много преимуществ, включая более высокую скорость обмена и менее интенсивное использование центрального процессора системой. Большая часть работы по реализации в драйвере KMDF функциональности DMA выполняется прозрачно инфраструктурой KMDF. От драйверов обычно требуется только указать способности своих устройств и инициировать операции DMA.

В этой главе описываются основные понятия и терминология, применяемые для написания драйверов DMA под Windows. В ней также приводятся подробные сведения о реализации DMA в устройстве, которые необходимо знать, прежде чем приступить к написанию драйвера DMA.

Информация в этой главе применима только к драйверам KMDF для устройств, способных работать с DMA.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты</b>	
Инструмент Driver Verifier	Встроенный в Windows
<b>Образцы драйверов</b>	
Образец драйвера PLX9x5x	%wdk%\src\kmdf\Plx9x5x
<b>Документация WDK</b>	
Раздел "Handling DMA Operations in Framework-Based Drivers" <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80070">http://go.microsoft.com/fwlink/?LinkId=80070</a>
Раздел "DMA Verification (Driver Verifier)" <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=80073">http://go.microsoft.com/fwlink/?LinkId=80073</a>

## Базовые понятия и терминология DMA

DMA представляет собой метод обмена данными между устройством и системной памятью без участия центрального процессора. Вместо того чтобы самому копировать данные из одного места в другое, центральный процессор только инициирует операцию копирования,

<sup>1</sup> Работа с операциями DMA в драйверах, основанных на инфраструктуре. — *Пер.*

<sup>2</sup> Верификация DMA (с помощью Driver Verifier). — *Пер.*

после чего предоставляет выполнение операции другим компонентам, а сам снова становится свободным для обслуживания других запросов. В зависимости от типа устройства DMA, фактическая операция копирования выполняется либо самим устройством, либо отдельным контроллером DMA. Конструкция таких устройств, как сетевые платы, обычно предоставляет поддержку DMA, что повышает производительность как самого устройства, так и всей системы.

Windows поддерживает два типа устройств DMA: устройства-мастера шины (bus-master) и системные устройства, которые иногда называются ведомыми (slave) устройствами.

Современные шины, такие как, например, PCI Express, обычно поддерживают устройства-мастера шины, которые в настоящее время являются наиболее распространенным типом устройств DMA. Устройство-мстер шины содержит все необходимые электронные компоненты и логику, чтобы взять управления — или "захватить" (от англ. *master* — овладевать, захватывать, управлять) — шиной, к которой оно подключено, и выполнять обмен данными между своим буфером и системной памятью хоста. Драйверы для устройств-мастеров шины могут воспользоваться поддержкой, оказываемой инфраструктурой для DMA.

Системные устройства DMA являютсяrudimentами первоначальной архитектуры IBM PC и обычно поддерживаются шиной ISA. Для обмена данными эти устройства полагаются на контроллер DMA на материнской плате. KMDF не поддерживает системный DMA, т. к. относительно немногие современные устройства поддерживают системный DMA. Драйверы для устройств системного DMA должны использовать WDM.

### Примечание

Если не указывается иное, то в этой главе термин "DMA" обозначает только DMA на основе захвата шины.

## Транзакции DMA и передачи DMA

Критически важными для понимания DMA, устройств DMA и модели DMA Windows являются следующие два термина:

- ◆ *транзакция DMA* — законченная операции ввода/вывода с применением DMA, например, один запрос от приложения на чтение или запись;
- ◆ *передача DMA* — одна аппаратная операция по передаче данных из системной памяти устройству или в обратном направлении.

Когда драйвер KMDF получает запрос ввода/вывода, он создает объект транзакции DMA для представления этого запроса. В зависимости от объема данных запроса, транзакция DMA может содержать одну или несколько передач. Инфраструктура внутренне определяет, может ли устройство выполнить всю транзакцию за одну передачу. Если транзакция слишком большая, то инфраструктура разбивает ее на несколько операций передачи, в каждой из которых передается фрагмент запрошенных данных.

Таким образом, одна передача DMA всегда ассоциирована с одной транзакцией DMA, но одна транзакция DMA может состоять из нескольких передач DMA.

## Пакетный DMA и DMA с применением общего буфера

Существуют два основных типа конструкции устройств DMA: пакетный DMA и DMA с применением общего буфера. Устройства также могут быть гибридной конструкции, содержащей элементы как пакетного DMA, так и DMA с применением общего буфера.

## Устройства DMA пакетной конструкции

Устройства DMA пакетной конструкции являются наиболее распространенными. В таких устройствах драйвер явно организовывает и выдает запрос на каждую передачу данных от устройства. Таким образом в каждой операции DMA передается один пакет данных.

Примером устройства пакетного DMA может служить устройство массовой памяти. Для считывания данных из такого устройства драйвер программирует устройство DMA для считывания требуемых секторов и предоставляет устройству указатель на буфер в системной памяти, в который поместить запрошенные данные. После завершения операции чтения устройство генерирует прерывание. По этому прерыванию драйвер выполняет необходимые операции по освобождению аппаратного обеспечения и завершает запрос.

## Устройства DMA с применением общего буфера

Для устройств DMA с применением общего буфера драйвер выделяет область в системной памяти (общий буфер), которую он использует совместно с устройством DMA. Формат этой области определяется устройством DMA и является понятным для драйвера. Эта общая область может содержать структуры данных, используемых драйвером для управления устройством, а также один или несколько буферов данных. Устройство периодически проверяет структуры данных в общей области и обновляет их, выполняя передачи DMA. Так как эти передачи DMA выполняются по требованию устройства, без каких-либо инициирующих действий со стороны драйвера, DMA с применением общего буфера иногда называется непрерывным (continuous) DMA.

Примером устройства DMA с общим буфером может служить интеллектуальный сетевой адаптер. Общий буфер такого адаптера может содержать набор буферов данных и пару циклических списков: один для приема, а другой для передачи. Каждый элемент в списке приема может содержать описание соответствующего буфера данных (в котором может содержаться входящее сетевое сообщение) и поле статуса, указывающее, является ли буфер данных доступным или же он уже содержит сообщение.

## Устройства DMA гибридной конструкции

Устройства DMA гибридной конструкции содержат элементы как пакетного DMA, так и DMA с применением общего буфера. Такие устройства обычно используют как расположенный на стороне хоста общий буфер памяти, содержащий структуры управления для устройства, так и набор описателей, содержащих информацию о каждом пакете данных, который требуется передать. Перед каждой передачей DMA программа организовывает описатели, после чего заносит в общий буфер запись о новом пакете, который нужно передать. Затем устройство передает пакет непосредственно из буфера данных, что составляет пакетную часть гибридной конструкции.

В отличие от конструкции чисто с применением общего буфера, драйвер не копирует содержимое буфера данных в общий буфер. По завершению передачи данных устройство выдает прерывание. Получив прерывание, драйвер устройства определяет, какие пакеты завершены, выполняет необходимые операции по освобождению аппаратного обеспечения и завершает соответствующие запросы.

## Поддержка метода "разбиение/объединение"

При использовании метода DMA "разбиение/объединение" (scatter/gather) в одной передаче содержатся данные из нескольких физических областей памяти. Некоторые устройства под-

держивают DMA типа "разбиение/объединение" на аппаратном уровне. Для устройств, чье аппаратное обеспечение не обладает такими возможностями, Windows реализует внутренний программный механизм разбиения/объединения.

Аппаратная поддержка для DMA типа "разбиение/объединение", также называемая *цеплением DMA* (DMA chaining), особенно эффективна в системах под управлением Windows. Так как в Windows применяется виртуальная память, с подкакой страниц по требованию, буферы данных, которые выглядят непрерывными в виртуальной памяти, в действительности, т. е. в физической памяти, не являются таковыми. Другими словами, непрерывной области виртуальной памяти буфера (объединение) на самом деле могут соответствовать разбросанные области физической памяти (разбиение). Устройство, поддерживающее DMA типа "разбиение/объединение" на аппаратном уровне, может завершить запрос за одну передачу, тогда как устройству, способному выполнять передачи DMA только из одной непрерывной области физической памяти, может потребоваться выполнить несколько меньших передач, чтобы завершить запрос.

Буфер для DMA типа "разбиение/объединение" описывается списком разбиения/объединения, который представляет собой просто перечень базовых адресов и размера каждой области физической памяти в буфере. Число элементов списка зависит от устройства, но для большинства современных устройств максимальное число элементов неограничено.

В драйверах KMDF не требуется реализовывать никакой специальной обработки для поддержки метода DMA "разбиение/объединение". Во время инициализации драйвер заполняет структуру данных конфигурации, описывающую возможности устройства, а инфраструктура потом применяет аппаратные возможности метода "разбиение/объединение" устройства или программные возможности Windows, в зависимости от того, что является уместным в конкретном случае.

## Информация об устройстве, специфичная для DMA

Разработка драйверов для устройств DMA является более сложной задачей, чем разработка драйверов для большинства других типов устройств. Эта задача может быть облегчена, если, прежде чем приступить к написанию драйвера, вы получите четкое понимание конструктивных особенностей устройства и то, каким образом драйвер может управлять им. В табл. 17.1 приведена информация, которую следует принять во внимание при проектировании драйвера.

**Таблица 17.1. Информация об устройстве, специфичная для DMA**

Информация об устройстве	Возможные значения
Тип конструкции DMA	Пакетный, с применением общего буфера или гибридный
Максимальная возможность адресации	32-, 64-битная или другая
Аппаратная поддержка метода "разбиение/объединение"	Да или нет
Максимальный размер передачи за одну операцию DMA	Количество байтов
Требуемое выравнивание буфера	Никакого, по границе слова, длинного слова (longword) или квадраслова (quadword)

Информация, приведенная в табл. 17.1, играет важную роль в принятии решений о том, какой должна быть общая конструкция ввода/вывода драйвера, в выборе профиля DMA для устройства и в конфигурировании объекта включателя DMA и объекта транзакции, используемых драйвером для выполнения DMA.

Так как существуют большие различия между реализациями устройств, то не обязательно все аспекты, приведенные в табл. 17.1, могут быть применимы к вашему драйверу. Кроме этого, все потенциальные проблемы, которые могут возникнуть в процессе разработки драйвера, рассмотреть просто невозможно. По вопросам, которые вы считаете уникальными для вашего устройства, проконсультируйтесь с разработчиком устройства или с каким-либо разработчиком аппаратуры, который понимает особенности конструкции вашего устройства.

## Информация об устройстве и конструкция драйверов DMA

В этом разделе детально рассматривается информация, принимаемая во внимание при разработке драйверов (см. табл. 17.1). В частности, рассматривается, каким образом специфичные особенности устройства могут повлиять на конструкцию драйвера DMA.

### Тип конструкции DMA

Наиболее важным из того, что необходимо знать об устройстве, является тип используемой в нем конструкции DMA — пакетная, с применением общего буфера или гибрид этих двух. Тип конструкции DMA определяет общую архитектуру и конструкцию драйвера.

Разработка драйверов для гибридных устройств и устройств с применением общего буфера обычно является более трудной задачей, чем для устройств с пакетной DMA. Причиной этому является то, что для разработки драйверов для устройств DMA с применением общего буфера требуется совершенное понимание и тщательная координация структур данных, используемых совместно драйвером и устройством. Некоторые вопросы, которые, возможно, необходимо выяснить, включают следующие:

- ◆ Каким образом драйвер сообщает устройству базовый адрес общего буфера?
- ◆ Какой формат применяется для указателей, сохраненных в общем буфере, например, как для связных списков?
- ◆ Какие требуются типы синхронизации между драйвером и устройством при обновлении структур в области общего буфера?

Конструкции драйверов, использующих общий буфер, могут вносить трудноопределимые побочные эффекты, с которыми драйвер должен будет в состоянии справиться. Например, устройства с общим буфером выполняют многократные операции обмена DMA с системной памятью при сканировании аппаратным обеспечением устройства разделяемых структур данных на изменения в них. Хотя это не является проблемой для большинства настольных систем, для портативных компьютеров такая конструкция скажется отрицательно на времени жизни аккумуляторной батареи. Если в портативном компьютере невозможно избежать использования конструкции устройства DMA с применением общего буфера, то обычно для такого устройства необходимо обеспечить реализацию драйвером активной схемы управления энергопотреблением. Например, конструкция драйвера может предоставлять возможность перевода устройства в состояние пониженного энергопотребления при его простое.

Некоторые гибридные конструкции допускают использование устройства как полностью пакетного. Так как разработка драйверов для устройств с пакетной DMA обычно является

менее сложной задачей, чем для устройств DMA с применением общего буфера, то при разработке драйвера для такого гибридного устройства можно сначала реализовать пакетный интерфейс, а поддержку для DMA с применением области общего буфера добавить позже.

## **Возможности адресации устройства**

Другим важным аспектом конструкции драйвера DMA-устройства является возможность физической адресации устройства, в частности, поддерживает ли устройство 64-битную адресацию. При выборе профиля DMA для устройства указывается, поддерживает ли оно передачи DMA по 64-битным адресам.

## **Аппаратная поддержка метода "разбиение/объединение"**

Современные устройства для работы под Windows должны иметь аппаратную поддержку метода "разбиение/объединение", т. к. это может значительным образом понизить накладные расходы и задержки при передаче данных. При выборе профиля DMA указывается, предоставляет ли устройство аппаратную поддержку метода "разбиение/объединение".

## **Максимальный объем данных передачи**

Для многих устройств имеется ограничение на максимальное количество байтов, которые могут быть переданы за одну операцию DMA. Если устройство, для которого разрабатывается драйвер, имеет такое ограничение, необходимо знать этот предел на объем передаваемых данных. Некоторые устройства такого ограничения на объем передачи DMA не имеют. Для таких устройств необходимо определить практический максимальный объем передачи, который разрабатываемый драйвер будет способным поддерживать.

Независимо от того, кем определяется максимальный объем передачи — устройством или драйвером — это значение необходимо предоставить при конфигурировании объекта включателя DMA.

## **Требуемое выравнивание буфера**

Для устройств DMA часто требуется, чтобы буферы данных, используемые в передачах DMA, были выровнены определенным образом. Например, для одного относительно распространенного чипсета для поддержки DMA требуется, чтобы буферы данных для операций чтения и записи DMA были выровнены по границе 16 байт. Причиной этого требования является то, что чипсет резервирует самые младшие 4 бита для своего собственного пользования. Как и с 64-битной адресацией и аппаратной поддержкой метода "разбиение/объединение", требование относительно выравнивания буфера помогает определить соответствующий профиль DMA для устройства.

## **Факторы, не принимающиеся к рассмотрению**

В Windows реализована обширная системная поддержка DMA в соответствии с абстракцией DMA в Windows, которая описывается далее в этой главе. Драйверы, применяющие для DMA методы KMDF, гарантированно соответствуют модели DMA Windows и, таким образом, для них можно избежать написания значительных объемов сложного кода для решения таких проблем, как следующие.

- ◆ **Поддержка 64-битной адресации.** Для драйверов, в которых применяется поддержка DMA, встроенная в KMDF, обычно не требуется реализовывать какой-либо специальный

код для правильной работы в 64-битном виртуальном адресном пространстве, предоставляемом 64-битными версиями Windows. Драйверы всегда должны использовать 64-битные безопасные типы данных, такие как, например, `ULONG_PTR`, чтобы определение размера их данных было ясным и однозначным. Драйверы, которым требуется различать между 32- и 64-битными вызывающими клиентами, должны вызывать для этого метод `WdfRequestIsFrom32BitProcess`.

- ◆ **Объем имеющейся физической памяти.** Возможности устройства определяют размер указателей на память, используемых инфраструктурой, но не возможности физической адресации или объем памяти, доступный на данной системе. Инфраструктура обеспечивает, чтобы физические адреса буферов данных, получаемых драйверами, всегда были в пределах возможностей адресации их устройств. Таким образом, драйверу для устройства, поддерживающего только 32-битную адресацию, не будет передан указатель, имеющий больше чем 32 значащих бита.
- ◆ **Возможности адресации шины.** Драйверам, использующим поддержку DMA, встроенную в KMDF, никогда не требуется определять возможности адресации шины, к которой они подключены, т. к. всеми вопросами адресации занимается инфраструктура. Например, драйверу для устройства шины PCI, поддерживающего 64-битную адресацию, не требуется знать, подключено ли устройство к шине PCI, поддерживающей 64-битную адресацию. Как и в случае с имеющимися в системе объемом физической памяти, возможности адресации шины делаются прозрачными инфраструктурной реализацией DMA.

Прозрачность адресации шины поддерживается для шин PCI с 64 линиями адреса, а также для шин PCI, поддерживающих режим DAC (Dual Address Cycle, двойной цикл адреса) 64-битной адресации.

Перечисленные возможности доступны только драйверам, использующим поддержку DMA, встроенную в KMDF, что соответствует модели реализации, основанной на абстракции DMA Windows. В зависимости от конкретных применяемых в них компонентов модели DMA, драйверы, в которых применяются решения по кратчайшему пути в обход модели — часто в неоправданном стремлении оптимизировать производительность — получают неопределенный уровень поддержки перечисленных ранее возможностей.

## Абстракция DMA в Windows

Архитектура DMA Windows представляет абстрактный вид базового аппаратного обеспечения системы. Этот вид, называемый *абстракцией DMA Windows*, создается менеджером ввода/вывода, уровнем HAL, драйверами шины, а также всеми имеющимися драйверами фильтра шины. По необходимости, изготовители OEM могут модифицировать некоторые из этих компонентов для поддержки уникальных возможностей их оборудования. Тем не менее, в большинстве компьютеров под управлением Windows применяются стандартные конструкции аппаратного обеспечения, которые поддерживаются стандартной реализацией DMA Windows.

Поддержка DMA в KMDF основана на абстракции DMA Windows. Благодаря применению абстракции DMA Windows, инфраструктуре не требуется иметь дело с уникальными возможностями и организацией базовой аппаратной платформы. KMDF и абстракция DMA Windows определяют стабильную аппаратную среду, на которую драйверы могут полностью полагаться. В результате, драйверам не требуется выполнять дополнительные проверки при

компиляции или исполнении для поддержки разных базовых аппаратных платформ. Абстракция DMA Windows описывает систему со следующими основными характеристиками.

- ◆ Операции DMA происходят непосредственно из системной памяти и нельзя предполагать их координацию с содержимым кэша процессора. Поэтому задача сброса любых измененных данных из кэша процессора обратно в системную память перед каждой операцией DMA является ответственностью инфраструктуры.
- ◆ После окончания каждой операции DMA часть переданных данных может быть оставлена в кэше операционной системой Windows или одним из чипсетов вспомогательной аппаратуры. В результате, инфраструктура является ответственной за сброс такого внутреннего кэша после каждой операции DMA.
- ◆ Адресное пространство любой шины устройств является отдельным от адресного пространства системной памяти. Преобразования между логическими адресами шины и физическими адресами системной памяти выполняются регистрами отображения (map registers). Это отображение позволяет программную поддержку механизма "разбиение/объединение" абстракцией DMA Windows.

Кроме этого, применение регистров отображения обеспечивает, что операции DMA на любойшине устройств могут достичь любого адреса в системной памяти. Например, от инфраструктуры не требуется выполнять никакой специальной обработки, чтобы нашине PCI с 32-битной адресацией обеспечить доставку операций DMA буферам данных, расположенных в системной памяти выше физического адреса 4 Гбайт (0xFFFFFFFF).

Абстракция DMA Windows состоит из нескольких ключевых областей:

- ◆ операции DMA и кэш процессора;
- ◆ завершение передач DMA сбросом кэшей;
- ◆ регистры отображения;
- ◆ системная поддержка механизма "разбиение/объединение";
- ◆ передача DMA по любому адресу физической памяти.

### **Полезная информация**

Для драйверов KMDF большинство детальностей по поддержке DMA обрабатывается инфраструктурой, поэтому информация в этом разделе не является критической для разработки драйверов. Но хотя познание всех деталей абстракции не является обязательным для написания драйверов KMDF, поддерживающих DMA, уверенное понимание базовых понятий может помочь в разработке и отладке драйвера.

## **Операции DMA и кэш процессора**

В абстракции DMA Windows операции DMA обходят системный аппаратный кэш и выполняются непосредственно в системной памяти. Поэтому, перед выполнением каждой операции DMA, инфраструктура должна обновить системную память, сбросив любые данные, находящиеся в аппаратном кэше системы, в системную память. Когда драйвер запрашивает передачу DMA, перед исполнением передачи инфраструктура сбрасывает данные.

Для реализации абстракции Windows обновляет системную память, только когда необходимо. На многих системах операции DMA правильно отображают содержимое аппаратного кэша системы, когда оно отличается от содержимого системной памяти. В результате, выполнение обратной записи (write-back) из кэша в системную память перед операцией DMA

не является необходимым. В таких системах стандартная реализация DMA Windows в действительности не выполняет никаких операций в ответ на запрос на сброс перед операцией DMA.

В системах или конфигурациях шины, в которых аппаратура не обеспечивает непротиворечивость кэшей для операций DMA (что имеет место в некоторых системах Intel Itanium), стандартная реализация DMA Windows выполняет специфичную для процесса работу, необходимую для обеспечения такой непротиворечивости кэшей, когда драйвер вызывает метод `WdfDmaTransactionExecute`.

## Завершение передач DMA сбросом кэшей

В соответствии с абстракцией DMA Windows, после завершения передачи DMA Windows или любой другой аппаратный компонент системы может оставить данные в кэше. Это кэширование является полностью прозрачным как для инфраструктуры, так и для драйвера. Поэтому, после выполнения каждой операции DMA, чтобы завершить операцию DMA, инфраструктура должна сбросить эти данные из внутреннего кэша Windows. Инфраструктура сбрасывает этот кэш, когда драйвер вызывает метод `WdfDmaTransactionDmaCompleted`.

Для реализации абстракции инфраструктура извещает Windows по завершении каждой передачи DMA. Это извещение позволяет операционной системе завершить должным образом все передачи, в которых используются регистры отображения. Подробное рассмотрение регистров отображения приводится в следующем разделе.

## Регистры отображения

Одной из основных особенностей абстракции DMA Windows является использование *регистров отображения* для преобразования адресов между системным адресным пространством и логическим адресным пространством, используемым шиной устройства. В этом разделе дается концептуальная абстракция регистров отображения, применяемых в модели DMA Windows, а также вкратце рассматривается воплощение регистров отображения в стандартной реализации DMA Windows.

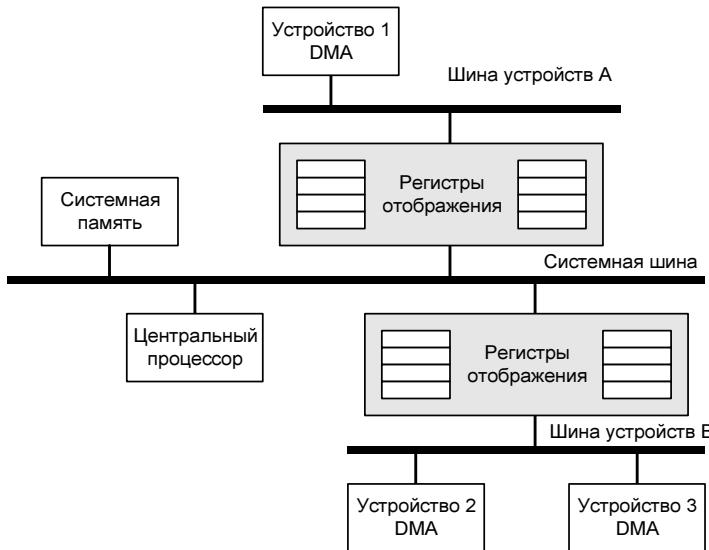
## Концепция

На рис. 17.1 приведена схема соединения системной памяти и шин устройств в абстракции DMA Windows. Для примера, шины устройств, показанные на схеме, могли бы быть шинами PCI. Но не забывайте, что рис. 17.1 не представляет организацию аппаратного обеспечения ни для какой конкретной системы, а является всего лишь общей концептуальной схемой.

На рисунке показаны две шины устройств, каждая из которых изолирована от системной шины. В абстракции DMA Windows каждая шина устройства обладает адресным пространством, отдельным как от адресного пространства всех других шин устройств, так и от адресного пространства системной памяти. Об адресном пространстве шины устройств говорят как о логическом адресном пространстве шины устройств для данной шины.

На рис. 17.1 каждая шина устройств подключена к шине памяти с помощью компонента, обозначенного "*регистры отображения*". Так как логическое адресное пространство шины устройств и адресное пространство системной памяти являются отдельными адресными пространствами, для преобразования их адресов требуется какой-то компонент. Таким компонентом и является группа регистров отображения. Регистры отображения преобразуют

адреса системной шины в адреса шины устройств, таким образом позволяя обмен данными между этими двумя шинами.



**Рис. 17.1.** Концептуальное представление связи шины устройств и системной шины памяти с помощью регистров отображения

Регистры отображения преобразуют адреса по большому счету таким же образом, как и регистры управления системной памятью (элементы каталога страниц и элементы таблицы страниц) выполняют преобразования между виртуальными адресами и адресами физической памяти. Каждый регистр отображения может преобразовать до одной страницы адресов в одном направлении. Размер страницы составляет 4 Кбайт на системах x86 и x64 и 8 Кбайт на системах Itanium и определяется в виде константы `PAGE_SIZE`.

Регистры отображения являются разделяемым ресурсом, которым управляет операционная система Windows. Система резервирует регистры отображения на основе максимального размера передачи, указанного драйвером при создании объекта включателя DMA. Инфраструктура выделяет регистры отображения непосредственно перед исполнением транзакции и освобождает их после завершения транзакции. Когда драйвер устройства DMA выполняет обмен данными с системной памятью, инфраструктура выделяет и программирует регистры отображения для этого обмена.

Так как каждый регистр отображения может преобразовать страницу адресов, количество регистров отображения, требуемых для передачи, зависит от размера передачи. Формула для вычисления точного количества регистров отображения следующая:

$$\text{Число требуемых регистров отображения} = (\text{Размер передачи} / \text{PAGE\_SIZE}) + 1.$$

В этой формуле дополнительный регистр отображения (+1) требуется на случай, когда передача не начинается с адреса, выровненного по границе страницы.

Регистры отображения выделяются в непрерывном блоке. Каждый блок регистров отображения представляет набор непрерывных логических адресов шины устройств, который может преобразовать набор физических адресов системной памяти такого же размера в любом направлении передачи. Для инфраструктуры KMDF Windows представляет блок регистров

отображения посредством значений базового адреса и длины блока. Для драйверов KMDF эти значения недоступны.

## Реализация

Так как регистры отображения являются концептуальной абстракцией, разработчики аппаратного обеспечения системы и работающие совместно с ними разработчики режима ядра могут реализовать их любым выбранным ими способом. На протяжении существования Windows было воплощено несколько интересных реализаций аппаратных регистров отображения. В некоторых из этих разработок для реализации преобразования между адресами шины устройств и адресами шины системной памяти применялась логика управления памятью подобно способу, показанному на рис. 17.1.

В большинстве современных компьютерных систем под управлением Windows применяется аппаратное обеспечение стандартной конструкции, и поэтому в них используется стандартная реализация DMA Windows. В этой реализации регистры отображения реализованы полностью программным способом.

Компоненты стандартного Windows DMA реализуют каждый регистр отображения как один буфер размером `PAGE_SIZE`, расположенный в физической памяти ниже границы в 4 Гбайт. В зависимости от шины, для которой организуется поддержка, регистры отображения могут располагаться в физической памяти даже ниже границы в 16 Мбайт. Когда драйвер выполняет подготовку к передаче DMA, Windows определяет, нужны ли регистры отображения для поддержки этой передачи, и если нужны, то сколько. Регистры отображения могут требоваться для поддержки передачи всего буфера данных или же для передачи только некоторых его фрагментов.

Если для осуществления передачи регистры отображения не требуются, то Windows предоставляет инфраструктуре физический адрес памяти фрагмента буфера в качестве логического адреса шины устройства для фрагмента.

Если для осуществления передачи требуются регистры отображения, то Windows выделяет область памяти для буфера на время передачи. Если для буфера нет достаточно памяти, то операционная система задерживает выполнение запроса до тех пор, пока не будет достаточно памяти для буфера. По завершению передачи DMA с использованием регистров отображения, Windows освобождает регистры отображения, таким образом делая память буфера доступной для использования в другой передаче.

## Когда применять регистры отображения

Концептуально, абстракция DMA Windows всегда применяет регистры отображения для преобразования между логическими адресами шины устройства и физическими адресами памяти.

Но при реализации концепции компоненты стандартного DMA Windows используют регистры отображения при наличии любого из следующих условий:

- ◆ драйвер устройства DMA указывает, что аппаратный механизм "разбиение/объединение" не поддерживается устройством;
- ◆ любая часть буфера, используемого для данной передачи, превышает возможности адресации устройства. Например, если устройство DMA может выполнять передачи только с 32-битной адресацией, но часть передаваемого буфера расположена выше границы физической памяти в 4 Гбайт, то система применяет регистры отображения.

## Поддержка системного механизма "разбиение/объединение"

Применение регистров отображения позволяет осуществить специальную поддержку для устройств, не реализующих аппаратную поддержку механизма "разбиение/объединение". В этом разделе описываются понятия, лежащие в основе поддержки системного механизма "разбиение/объединение" в Windows, а также каким образом компоненты стандартной DMA Windows реализуют поддержку системного механизма "разбиение/объединение".

### Концепция

В системах Windows непрерывные буферы данных, скорее, исключение, нежели правило. Если буферы не являются непрерывными, то для осуществления DMA для устройства, не имеющего аппаратной поддержки механизма "разбиение/объединение", может легко потребоваться выполнить большой объем дополнительной обработки, т. к. для каждого физического фрагмента пользовательского буфера потребуется отдельная передача DMA. Но поддержка системного механизма "разбиение/объединение" решает проблему дополнительной обработки.

Регистры отображения, которые преобразуют логические адреса шины в физические адреса системной памяти и обратно, выполняют все преобразования адресов при обмене данными междушиной устройств ишиной системной памяти. Для реализации программной поддержки механизма "разбиение/объединение" Windows выделяет непрерывные регистры отображения для данной передачи. В результате передачи, в которых используются регистры отображения, для устройства всегда выглядят как набор смежных логических адресов шины устройств, даже если соответствующие страницы системной памяти не являются физически смежными.

Для наглядной демонстрации концепции на рис. 17.2 показано, как последовательность смежных регистров отображения описывает фрагментированный буфер данных. Компоненты стандартной DMA Windows запрограммировали регистры отображения указывать на разные области в физической памяти хоста. Но логические адреса шины устройств, представляемые регистрами отображения, расположены в смежном порядке, поэтому система может использовать лишь один базовый логический адрес шины устройств и размер блока адресов, чтобы описать их для устройства DMA.

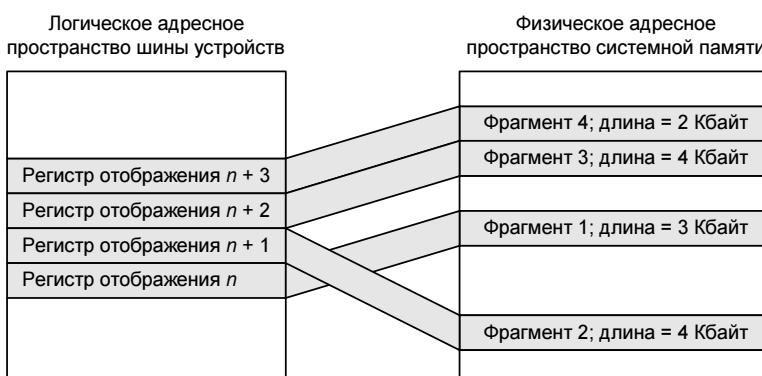


Рис. 17.2. Преобразование фрагментированного физического буфера в непрерывный логический с помощью регистров отображения

Если у вас все еще есть трудности с пониманием этой абстракции, вспомните, что регистры отображения концептуально преобразуют физические адреса системной памяти в логические адреса шины устройств для передач DMA точно таким же образом, как системное аппаратное обеспечение управления памятью преобразует виртуальные адреса в физические для исполнения программы.

## Реализация

Регистры отображения поддерживают системный механизм "разбиение/объединение" посредством промежуточной буферизации передач DMA. Во время инициализации, если драйвер указывает, что его устройство не предоставляет аппаратной поддержки механизма "разбиение/объединение", то реализация DMA Windows использует регистры отображения для всех обменов DMA с устройством. Таким образом, для таких устройств передача DMA выполняется следующим образом:

1. Реализация DMA Windows выделяет достаточное количество смежных регистров отображения (т. е. буферов в нижней памяти), чтобы в них поместились все данные передачи. Если нет достаточно памяти для буферов, то операционная система задерживает выполнение запроса до тех пор, пока не будет достаточно памяти для буферов.
2. Для операции записи (т. е. передачи из системной памяти в устройство) реализация DMA копирует данные из первоначального буфера данных в буфер регистров отображения.
3. Реализация DMA Windows предоставляет инфраструктуре адрес физической памяти буфера регистров отображения в качестве логического адреса шины устройств для буфера.

После этого инфраструктура передает этот адрес драйверу в списке "разбиение/объединение", когда она вызывает функцию обратного вызова *EvtProgramDma* драйвера для программирования устройства для передачи DMA. Так как буфер является физически непрерывным, то список "разбиение/объединение" содержит только один элемент: базовый адрес и размер буфера. Драйвер использует этот адрес и размер буфера (а не настоящие адреса фрагментов буфера данных) для программирования устройства для операции DMA. Кроме этого, т. к. буферы регистров отображения находятся в диапазоне физической памяти, верхняя граница которого равна 4 Гбайт, то такие буферы всегда будут в пределах возможностей адресации любого устройства DMA мастера шины.

Когда драйвер извещает инфраструктуру о завершении передачи DMA, инфраструктура в свою очередь извещает стандартную реализацию DMA Windows, которая определяет, использовались ли при передаче регистры отображения и являлась ли операция операцией чтения из устройства в системную память.

При положительном результате проверки Windows копирует содержимое буферов регистров отображения, использованных в передаче, в исходный буфер данных.

После этого система освобождает регистры отображения, позволяя использовать их для других передач DMA.

## Передача DMA по любому адресу физической памяти

Используя регистры отображения, абстракция DMA Windows может поддерживать передачи любого размера, независимо от возможностей адресации шины устройств и объема системной памяти. В результате, в Windows даже устройства, имеющие ограниченные возможности адресации, могут обращаться ко всей физической памяти.

В этом разделе описывается, каким образом регистры отображения позволяют устройству DMA передавать данные по любому адресу физической памяти.

## Концепция

Допустим, что устройство DMA 1 на рис. 17.1 является мастером шины на шине устройств А. Так как устройство 1 способно выполнять только 32-битную адресацию, оно может предоставить на шину устройств только адреса в диапазоне от 0x00000000 до 0xFFFFFFFF. Другими словами, оно может адресовать 4 Гбайт памяти. Если бы шина устройства А была подключена к системной памяти напрямую, то она могла бы обмениваться данными только с нижними 4 Гбайт физического пространства системы. Организация работы системы виртуальной памяти Windows не может предотвратить размещение программой, использующей устройство 1, своих буферов данных выше границы 4 Гбайт в системе, имеющей свыше 4 Гбайт физической памяти. Более того, это невозможно на любой системе, в которой память расположена выше отметки 4 Гбайт физического пространства. Чтобы решить эту проблему, в передачах DMA устройства 1 нужно было бы или использовать буфера, специально выделенные ниже границы 4 Гбайт, или же реализовать специальную обработку для физических адресов любых фрагментов буферов данных пользователя, находящихся вне диапазона адресации устройства.

Абстракция DMA Windows позволяет избежать такой специальной обработки. В рамках организации передачи DMA, функции DMA Windows выделяют и программируют регистры отображения между шиной устройств А и шиной системной памяти для выполнения необходимого перемещения. Регистры отображения преобразуют 32-битные логические адреса шины устройств в 36-битные физические адреса шины системной памяти таким же образом, как регистры управления памятью некоторых систем x86 преобразуют 32-битные виртуальные адреса в 36-битные физические адреса.

## Реализация

В стандартной реализации DMA Windows регистры отображения воплощены полностью программным образом в виде смежных буферов размером PAGE\_SIZE в системной памяти по физическому адресу ниже 4 Гбайт.

При каждой организации передачи DMA драйвером KMDF компоненты Windows DMA определяют потребность в регистрах отображения, проверяя возможности адресации устройства и расположение передаваемого буфера данных. Если какая-либо часть буфера находится вне пределов возможности физической адресации устройства, система привлекает к выполнению передачи регистры отображения. Таким образом, если в передаче, организованной 32-битным устройством DMA, используется буфер данных, содержащий один или несколько фрагментов, расположенных выше физической границы 4 Гбайт, то для выполнения такой передачи требуется задействовать регистры отображения.

Windows использует регистры отображения только для фрагментов буфера данных, расположенных вне диапазона адресации устройства. Если весь фрагмент расположен в памяти, которую устройство способно адресовать, то Windows предоставляет физический адрес памяти фрагмента в качестве логического адреса шины устройств для этого фрагмента.

Если фрагмент находится вне пределов возможностей адресации устройства, стандартная реализация DMA Windows осуществляет передачу следующим образом:

1. Выделяет регистр отображения для хранения данных фрагмента. Если нет достаточно памяти для буферов, то операционная система задерживает выполнение запроса до тех пор, пока не будет достаточно памяти для буферов.

2. Для операции записи (т. е. передачи из системной памяти в устройство) реализация DMA копирует данные из первоначального буфера данных в буфер регистра отображения.
3. Предоставляет физический адрес памяти буфера регистра отображения в качестве логического адреса шины устройств для фрагмента.

Инфраструктура использует физические адреса памяти, предоставленные системой для создания списка "разбиение/объединение". Каждый элемент в этом списке представляет физический адрес памяти или фрагмента буфера данных или регистра отображения. Когда инфраструктура вызывает функцию обратного вызова *EvtProgramDma* драйвера, она передает ей этот список в качестве параметра. Драйвер использует элементы списка "разбиение/объединение", чтобы запрограммировать свое устройство для выполнения операции DMA.

Когда драйвер и инфраструктура извещают о завершении передачи DMA, стандартная реализация DMA Windows определяет, были ли во время передачи использованы регистры отображения. Если регистры отображения применялись и если выполнялась операция чтения из устройства в системную память, стандартная реализация DMA Windows копирует содержимое всех буферов регистров отображения, использованных в передаче, в исходный буфер данных. После этого Windows освобождает регистры отображения, позволяя использовать их для других передач DMA.

## Реализация драйверов DMA

Для поддержки DMA в драйвере KMDF требуется код в нескольких функциях обратного вызова для событий драйвера, как показано на рис. 17.3.

Как можно видеть на рис. 17.3, обработка, связанная с DMA, выполняется в четыре шага:

1. Во время инициализации драйвера, обычно в функции обратного вызова *EvtDriverDeviceAdd*, драйвер инициализирует и создает объект выключателя DMA и объект общего буфера, которые требуются для поддержки устройства DMA.
2. Когда драйвер получает запрос ввода/вывода, требующий DMA, если объект транзакции еще не был создан, то драйвер создает такой объект и инициирует транзакцию DMA. Этот код обычно находится в функции обратного вызова *EvtIoRead* или *EvtIoWrite* или в другой функции обратного вызова для события ввода/вывода. Но он также может быть в другой функции драйвера, если драйвер организовал свою очередь для ручной диспетчеризации.
3. Организовав буфера, требуемые для выполнения передачи, инфраструктура вызывает функцию обратного вызова *EvtProgramDma* драйвера. Эта функция программирует аппаратное обеспечение устройства для выполнения передачи DMA.
4. При каждом завершении передачи DMA аппаратным обеспечением драйвер выполняет проверку на завершение транзакции, обычно делая это в функции обратного вызова *EvtInterruptDpc*. При положительном результате проверки драйвер завершает запрос ввода/вывода. В противном случае, инфраструктура подготавливает следующую передачу и повторяет шаг 3.

Каждый из этих шагов рассматривается в подробностях в следующих разделах, с применением демонстрационного кода, адаптированного из образца драйвера KMDF PLX9x5x, поставляемого вместе с набором разработчика WDK. Данный образец драйвера поддерживает устройство PCI, оборудованное ресурсами порта, памяти, прерывания и DMA. Устройство

можно остановить и запустить при работающей системе; оно также поддерживает состояния пониженного энергопотребления. Аппаратное обеспечение имеет два канала DMA, что позволяет драйверу использовать отдельные каналы для операций чтения и записи. Драйвер конфигурирует две очереди с последовательной диспетчеризацией: одну для запросов на чтение, а вторую — для запросов на запись.

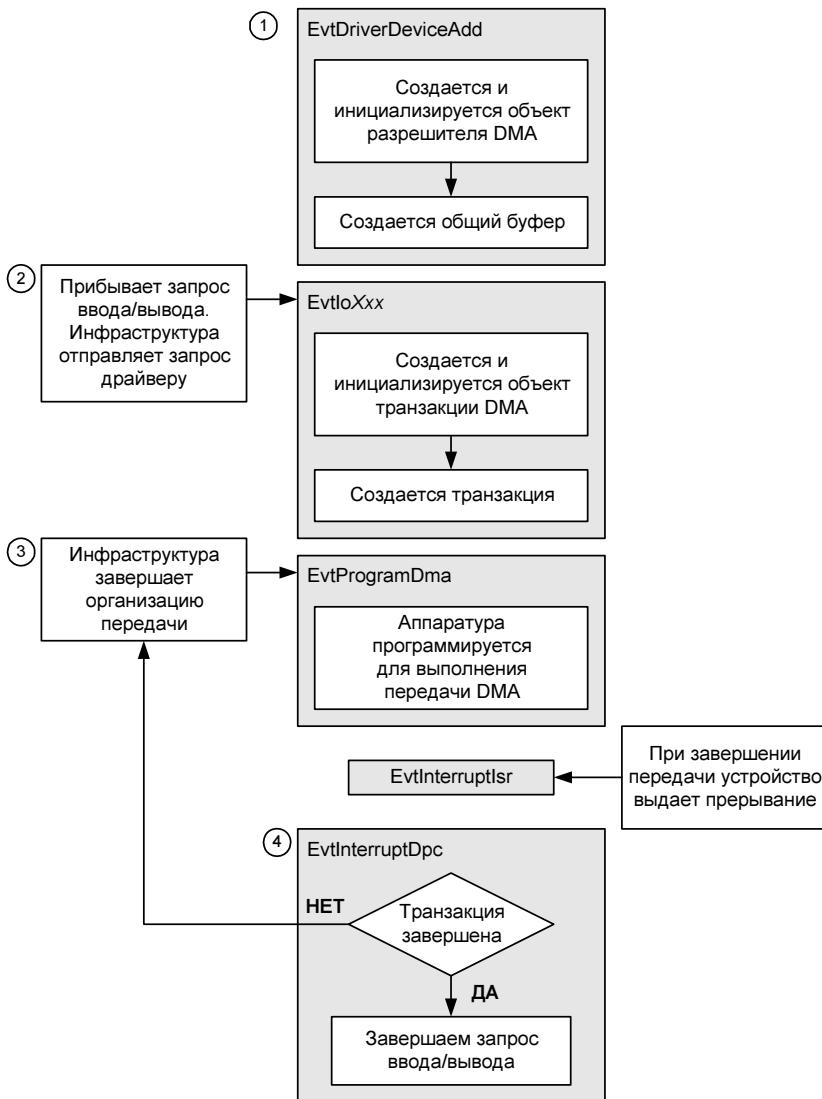


Рис. 17.3. Реализация DMA в драйверах KMDF

## Инициализации драйвера DMA

Во время инициализации, обычно в функции обратного вызова `EvtDriverDeviceAdd`, драйвер конфигурирует и создает объект выключателя DMA, и если устройство поддерживает DMA с применением общего буфера, то также и объект общего буфера.

Если драйвер указал пакетный DMA-профиль, он должен сериализировать все свои транзакции, т. к. инфраструктура позволяет исполнение только одной транзакции пакетной DMA за раз. Для реализации сериализации драйвер должен либо отправлять все запросы ввода/вывода, требующие DMA, из одной очереди с последовательной диспетчеризацией, либо реализовать ручную диспетчеризацию и вызывать инфраструктуру для получения следующего запроса ввода/вывода только после завершения предыдущего запроса.

Если для отправки запросов ввода/вывода DMA драйвер конфигурирует очередь с последовательной диспетчеризацией, то он также должен создать инфраструктурный объект транзакции DMA во время инициализации. В таком случае в любое время имеется только один активный запрос, поэтому драйвер может создать во время инициализации только один объект транзакции DMA и использовать его повторно для каждого запроса DMA.

## Объект выключателя DMA

Драйвер использует объект выключателя DMA (т. е. `WDFDMAENABLER`) для взаимодействия с инфраструктурой относительно передач DMA для конкретного объекта устройства. Объект выключателя DMA содержит информацию о возможностях DMA-устройства.

Прежде чем создать этот объект, драйвер должен сначала инициализировать структуру `WDF_DMA_ENABLER_CONFIG`, вызывая для этого макрос `WDF_DMA_ENABLER_CONFIG_INIT`. Данный макрос инициализирует структуру профилем DMA и максимальным размером передачи для устройства.

Профиль DMA указывает основные характеристики DMA-устройства, такие как поддержка 64-битной адресации и аппаратного метода "разбиение/объединение". Профили DMA и соответствующие атрибуты приведены в табл. 17.2.

**Таблица 17.2. Профили DMA KMDF и их значения**

Наименование профиля	Поддержка 64-битной адресации	Аппаратная поддержка метода "разбиение/объединение"	Поддержка одновременных операций чтения и записи
<code>WdfDmaProfilePacket</code>	Нет	Нет	Нет
<code>WdfDmaProfileScatterGather</code>	Нет	Да	Нет
<code>WdfDmaProfileScatterGatherDuplex</code>	Нет	Да	Да
<code>WdfDmaProfilePacket64</code>	Да	Нет	Нет
<code>WdfDmaProfileScatterGather64</code>	Да	Да	Нет
<code>WdfDmaProfileScatterGather64Duplex</code>	Да	Да	Да

Инициализировав структуру, драйвер создает объект выключателя DMA, вызывая для этого метод `WdfDmaEnablerCreate`.

## Объект общего буфера

Если устройство выполняет DMA с использованием общего буфера, то драйвер должен также создать объект общего буфера (т. е. `WDFCOMMONBUFFER`), вызывая для этого метод `WdfCommonBufferCreate` или `WdfCommonBufferCreateWithConfig`. Эти методы почти идентичны, за исключением того, что метод `WdfCommonBufferCreateWithConfig` принимает дополнитель-

ный входной параметр — структуру `WDF_COMMON_BUFFER_CONFIG`, в которой содержатся требования по выравниванию для буфера.

Дальше драйвер должен получить виртуальный адрес системы и логический адрес шины устройства, вызывая для этого следующие два метода WDF:

- ◆ `WdfCommonBufferGetAlignedVirtualAddress` — возвращает системный виртуальный адрес общего буфера;
- ◆ `WdfCommonBufferGetAlignedVirtualAddress` — возвращает логический адрес шины устройств для общего буфера.

Эти адреса требуются для программирования устройства в функции обратного вызова `EvtProgramDma`.

## Объект транзакции DMA

Запросы ввода/вывода, для которых образец драйвера выполняет DMA, отправляются из очереди с последовательной диспетчеризацией. Поэтому во время инициализации драйвер создает объект транзакции DMA. Вместо того чтобы создавать и удалять объект транзакции DMA для каждого нового запроса, драйвер может сохранить этот объект транзакции DMA и использовать его с последующими транзакциями DMA.

Для создания объекта DMA-транзакции драйвер вызывает метод `WdfDmaTransactionCreate`, передавая ему в качестве параметров созданный ранее объект выключателя DMA. Метод возвращает драйверу дескриптор созданного объекта транзакции DMA. Как инфраструктура, так и драйвер используют объект транзакции DMA для управления операциями DMA для данного запроса.

## Пример: инициализации драйвера DMA

В листинге 17.1 приведен пример инициализации драйвера для гибридного устройства. Таким образом, в нем демонстрируется выполнение необходимой инициализации для обеспечения поддержки как пакетной DMA, так и DMA с использованием общего буфера. Данный пример взят из файла `Sys\Init.c` образца драйвера `PLX9x5x`.

### Листинг 17.1. Инициализации драйвера DMA

```

WDF_DMA_ENABLER_CONFIG dmaConfig;
WdfDeviceSetAlignmentRequirement(DevExt->Device,
                                PCI9656_DTE_ALIGNMENT_16);
WDF_DMA_ENABLER_CONFIG_INIT(&dmaConfig,
                            WdfDmaProfileScatterGather64Duplex,
                            DevExt->MaximumTransferLength);
status = WdfDmaEnablerCreate(DevExt->Device,
                             &dmaConfig, WDF_NO_OBJECT_ATTRIBUTES,
                             &DevExt->DmaEnabler);
if (!NT_SUCCESS (status)) {
    . . . // Код для обработки ошибки опущен.
}
// Выделяем общий буфер для организации операций записи.
DevExt->WriteCommonBufferSize =
    sizeof(DMA_TRANSFER_ELEMENT) * DevExt->WriteTransferElements;

```

```
status = WdfCommonBufferCreate(DevExt->DmaEnabler,
                               DevExt->WriteCommonBufferSize,
                               WDF_NO_OBJECT_ATTRIBUTES,
                               &DevExt->WriteCommonBuffer);

if (!NT_SUCCESS(status)) {
    . . . // Код для обработки ошибки опущен.
}

DevExt->WriteCommonBufferBase = WdfCommonBufferGetAlignedVirtualAddress
                                (DevExt->WriteCommonBuffer);
DevExt->WriteCommonBufferBaseLA = WdfCommonBufferGetAlignedLogicalAddress
                                (DevExt->WriteCommonBuffer);

RtlZeroMemory(DevExt->WriteCommonBufferBase,
              DevExt->WriteCommonBufferSize);

WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes,
                                       TRANSACTION_CONTEXT);

status = WdfDmaTransactionCreate(DevExt->DmaEnabler,
                                 &attributes,
                                 &DevExt->ReadDmaTransaction);

if (!NT_SUCCESS(status)) {
    . . . // Код для обработки ошибки опущен.
}
```

Код, приведенный в листинге 17.1, выполняет следующие задачи:

1. Устанавливает требуемое выравнивание для объекта устройства.
2. Инициализирует и создает объект выключателя DMA.
3. Создает общий буфер.
4. Получает адреса общего буфера.
5. Создает объект транзакции DMA.

Первым делом образец драйвера устанавливает требуемое выравнивание для объекта устройства. Инфраструктура использует это значение как выравнивание для DMA, если драйвер не укажет требуемое выравнивание при создании общего буфера.

Далее драйвер инициализирует структуру `WDF_DMA_ENABLER_CONFIG`, вызывая для этого макрос `WDF_DMA_ENABLER_CONFIG_INIT`. Драйвер указывает профиль DMA, который описывает устройство наилучшим образом, а также максимальный поддерживаемый устройством размер передачи для одной операции DMA. Драйвер выбирает профиль `WdfDmaProfileScatterGather64`, чтобы указать, что устройство поддерживает как 64-битную адресацию, так и аппаратную реализацию метода "разбиение/объединение".

Инициализировав структуру `WDF_DMA_ENABLER_CONFIG`, драйвер создает объект выключателя DMA, вызывая для этого метод `WdfDmaEnablerCreate`. Драйвер сохраняет дескриптор созданного объекта для дальнейшего использования.

Данное устройство является гибридной конструкцией, т. е. оно поддерживает комбинацию пакетной DMA и DMA с использованием общего буфера, поэтому драйвер создает общий буфер. Для этого он вызывает метод `WdfCommonBufferCreate`, передавая ему в качестве параметра требуемый размер общего буфера в байтах. Выделенная область общего буфера не обязательно будет физически непрерывной. По умолчанию для общего буфера применяется такое же выравнивание, какое было указано ранее в вызове метода `WdfDeviceSetAlignmentRequirement`. Альтернативно, драйвер мог бы вызвать метод `WdfCommonBufferCreateWithConfig`, чтобы создать буфер и установить требования для выравнивания.

Кроме выделения памяти для общего буфера, методы `WdfCommonBufferCreate` и `WdfCommonBufferCreateWithConfig` также выделяют смежные регистры отображения в достаточном количестве для преобразования диапазона физических адресов общего буфера в логические адреса шины устройств. Эти методы также программируют эти регистры отображения для выполнения необходимых преобразований между логическими и физическими адресами шины устройств.

Далее драйвер вызывает метод `WdfCommonBufferGetAlignedVirtualAddress`, чтобы получить виртуальный адрес режима ядра для только что созданного им общего буфера. Драйвер использует этот адрес для манипулирования структурами данных в области общего буфера, которые он использует совместно с устройством. Драйвер завершает свою инициализацию, специфичную для DMA, вызывая метод `WdfCommonBufferGetAlignedLogicalAddress`, чтобы получить логический адрес шины устройств для общего буфера.

Наконец, драйвер создает объект транзакции DMA, вызывая метод `WdfDmaTransactionCreate` и передавая ему в качестве параметра дескриптор объекта включателя DMA. Драйвер использует этот объект транзакции для всех запросов DMA на чтение.

## Инициирование транзакции

Когда драйвер получает запрос ввода/вывода, требующий DMA, он инициирует транзакцию DMA. Код для этого обычно находится в функции обратного вызова `EvtIoRead` или `EvtIoWrite` или в другой функции обратного вызова для события ввода/вывода. Но он также может находиться в другом месте, если драйвер извлекает запросы ввода/вывода из очереди самостоятельно.

### Инициализация транзакции

Прежде чем драйвер может инициировать транзакцию DMA, он должен инициализировать объект транзакции DMA информацией о запрошенной передаче. Инфраструктура использует эту информацию совместно с профилем DMA, предоставленного драйвером в объекте включателя DMA, для вычисления числа требуемых регистров отображения и для создания списка "разбиение/объединение", используемого драйвером для программирования устройства.

Если драйвер еще не создал объект транзакции DMA для применения в этой транзакции, то сначала он должен создать этот объект, вызвав метод `WdfDmaTransactionCreate`.

После этого драйвер может инициализировать транзакцию, вызвав метод `WdfDmaTransactionInitializeUsingRequest` или `WdfDmaTransactionInitialize`.

Если драйвер получил запрос ввода/вывода от инфраструктуры, то для инициализации объекта транзакции данными из объекта запроса применяется метод `WdfDmaTransactionInitializeUsingRequest`. В качестве входных параметров этому методу передается указатель на объект `WDFREQUEST`, который следует обработать, константа перечисления, указывающая направление передачи — из устройства или в устройство, а также указатель на функцию обратного вызова `EvtProgramDma` драйвера.

Если драйвер выполняет DMA с использованием общего буфера или транзакции DMA, не исходящие из запроса ввода/вывода, для инициализации объекта транзакции вызывается метод `WdfDmaTransactionInitialize`. В качестве параметров методу передается константа направления передачи, указатель на функцию обратного вызова `EvtProgramDma`, а также указатель на список MDL, описывающий буфер для передачи, виртуальный адрес буфера и длина

буфера. Чтобы получить указатель на список MDL, для запросов на запись драйвер вызывает метод `WdfRequestRetrieveInputWdmMdl`, а для запросов на чтение — метод `WdfRequestRetrieveOutputWdmMdl`, после чего вызывает функции менеджера памяти ядра, чтобы получить его виртуальный адрес и длину.

## Исполнение транзакции

Инициализировав объект транзакции DMA, драйвер может начать обработку транзакции DMA, вызывая метод `WdfDmaTransactionExecute`. Перед началом исполнения транзакции DMA этот метод сбрасывает все измененные данные из кэша процессора обратно в системную память. После этого он вызывает функцию обратного вызова `EvtProgramDma` драйвера, чтобы запросить драйвер запрограммировать устройство для данной передачи DMA.

### Пример: иницирование транзакции

Следующий пример также основан на образце драйвера PLX9x5x. Код в листинге 17.2 показывает операции, выполняемые типичным драйвером KMDF для иницирования передачи DMA. Этот исходный код находится в файле Read.c.

#### Листинг 17.2. Иницирование транзакции DMA

```
VOID PLxEvtIoRead(IN WDFQUEUE Queue,
                   IN WDFREQUEST Request,
                   IN size_t Length)
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    PDEVICE_EXTENSION devExt;
    // Получаем DevExt с указателя в очереди
    devExt = PLxGetDeviceContext(WdfIoQueueGetDevice(Queue));
    do {
        // Проверяем достоверность параметра Length.
        if (Length > PCI9656_SRAM_SIZE) {
            Status = STATUS_INVALID_BUFFER_SIZE;
            break;
        }
        // Инициализируем транзакцию DMA.
        status = WdfDmaTransactionInitializeUsingRequest
            (devExt->ReadDmaTransaction,
             Request,
             PLxEvtProgramReadDma,
             WdfDmaDirectionReadFromDevice);
        if (!NT_SUCCESS(status)) {
            . . . // Код для обработки ошибки опущен.
            break;
        }
        // Исполняем данную транзакцию DMA.
        status = WdfDmaTransactionExecute(devExt->ReadDmaTransaction,
                                         WDF_NO_CONTEXT);
        if (!NT_SUCCESS(status)) {
            . . . // Код для обработки ошибки опущен.
            break;
        }
    }
```

```

// Указываем, что транзакция DMA запустилась успешно.
// Процедура DPC завершит запрос по завершению транзакции DMA.
status = STATUS_SUCCESS;
} while (0);
// Если имеются ошибки, выполняем очистку и завершаем запрос.
if (!NT_SUCCESS(status)) {
    WdfDmaTransactionRelease(devExt->ReadDmaTransaction);
    WdfRequestComplete(Request, status);
}
return;
}

```

В листинге 17.2 показана функция обратного вызова *EvtIoRead* драйвера, которая выполняет следующие операции для иницирования транзакции DMA:

1. Получает указатель на область контекста устройства, которая содержит дескриптор объекта транзакции DMA.
2. Проверяет достоверность длины передачи.
3. Инициализирует транзакцию.
4. Запускает транзакцию на исполнение.

Первым делом драйвер вызывает функцию доступа *PLxGetDeviceContext*, чтобы получить указатель на область контекста своего объекта *WDFDEVICE*. Потом он проверяет действительность длины передачи.

Дальше драйвер вызывает метод *WdfDmaTransactionInitializeUsingRequest*, чтобы ассоциировать запрос, переданный инфраструктурой его функции обратного вызова *EvtIoRead*, с созданным им ранее объектом транзакции DMA. В качестве входных параметров этому методу передается дескриптор объекта запроса ввода/вывода и дескриптор объекта транзакции DMA. Также ему передается указатель направления передачи — *WdfDmaDirectionReadFromDevice* или *WdfDmaDirectionWriteToDevice* — и указатель на функцию обратного вызова драйвера для события *EvtProgramDma*, которая называется *PLxEvtProgramDma*. Метод *WdfDmaTransactionInitializeUsingRequest* проверяет достоверность параметров для запроса и организует как можно больше внутренней инфраструктуры.

В случае успешного завершения метода *WdfDmaTransactionInitializeUsingRequest* драйвер вызывает метод *WdfDmaTransactionExecute*. Этот метод выполняет следующие действия:

1. Определяет размер передачи DMA.

Размер передачи DMA зависит от того, может ли текущий запрос быть удовлетворен в одной передаче или же транзакцию необходимо разбить на несколько передач. Причиной для разбиения транзакции на несколько передач могут быть накладываемые устройством ограничения на размер передачи или ограниченное количество доступных регистров отображения. Если инфраструктура может обработать весь запрос за одну передачу DMA, то она так это и делает. В противном случае инфраструктура разбивает транзакцию на несколько передач DMA и обрабатывает их в последовательном порядке.

2. Выдает запрос для Windows синхронизировать содержимое кэша процессора с системной памятью для целей передачи DMA.
3. Выделяет и инициализирует ресурсы, необходимые для выполнения передачи.

Этот шаг включает выделение и программирование всех необходимых регистров отображения и создание списка "разбиение/объединение", который будет передан драйверу.

4. Вызывает функцию обратного вызова *EvtProgramDma* драйвера, передавая ей указатель на созданный список пар "базовый логический адрес шины устройства/длина передачи", чтобы драйвер мог запрограммировать устройство для инициирования операции DMA.

Если во время инициирования происходит ошибка, то драйвер вызывает метод *WdfDmaTransactionRelease*, чтобы освободить ресурсы, задействованные инфраструктурой, без удаления объекта транзакции. После этого драйвер завершает запрос ввода/вывода обычным образом, возвращая статус ошибки.

Если метод *WdfDmaTransactionExecute* определит, что транзакцию DMA необходимо выполнять в нескольких передачах DMA, то инфраструктура исполняет эти передачи последовательным образом. То есть, инфраструктура определяет размер первой передачи и вызывает функцию обратного вызова *EvtProgramDma* драйвера, чтобы запрограммировать устройство для данной передачи DMA. Далее, когда первая передача завершена и драйвер вызывает метод *WdfDmaTransactionDmaCompleted* (обычно со своей функции *EvtInterruptDpc*), инфраструктура проверяет, была ли завершена вся транзакция. Если не была, то инфраструктура определяет размер следующей передачи и снова вызывает функцию обратного вызова *EvtProgramDma* драйвера для выполнения данной передачи. Этот цикл повторяется до тех пор, пока не будет завершена вся транзакция DMA. После этого драйвер может завершить соответствующий запрос ввода/вывода.

## Обработка запроса

Функция обратного вызова *EvtProgramDma* драйвера программирует устройство DMA для выполнения передачи. Инфраструктура передает драйверу указатель на список "разбиение/объединение", содержащий пары "логический адрес шины устройства/размер", которые совместно описывают передачу. Драйвер использует этот список пар "адрес/размер", чтобы запрограммировать устройство для выполнения передачи DMA.

### Определение функции *EvtProgramDma*

Далее приводится прототип функции обратного вызова *EvtProgramDma*:

```
typedef
BOOLEAN
(*PFN_WDF_PROGRAM_DMA) (
    IN WDFDMA TRANSACTION Transaction,
    IN WDFDEVICE Device,
    IN WDFCONTEXT Context,
    IN WDF_DMA_DIRECTION Direction,
    IN PSCATTER_GATHER_LIST SgList
);
```

Значения используемых в функции переменных таковы:

- ◆ *Transaction* — дескриптор объекта транзакции DMA, который представляет текущую транзакцию DMA;
- ◆ *Device* — дескриптор инфраструктурного объекта устройства;
- ◆ *Context* — указатель на контекст, который драйвер обозначил в предыдущем вызове метода *WdfDmaTransactionExecute*;
- ◆ *Direction* — константа перечисления *WDF\_DMA\_DIRECTION*, указывающая направление передачи DMA;
- ◆ *SgList* — указатель на структуру *SCATTER\_GATHER\_LIST*.

При успешном запуске передачи DMA функция *EvtProgramDma* должна возвратить TRUE и FALSE — в противном случае.

## Задачи, выполняемые функцией *EvtProgramDma*

После того как инфраструктура организует транзакцию, она вызывает функцию обратного вызова *EvtProgramDma* драйвера для выполнения передачи DMA. Эта функция должна выполнить следующие операции:

1. Определить смещение в буфере, с которого нужно начинать передачу.
2. Установить адреса и размеры для программирования устройства.
3. Запрограммировать устройство и начать передачу.
4. Освободить или удалить транзакцию в случае ошибки.

Не забывайте, что одна транзакция DMA может состоять из нескольких передач DMA. Такая ситуация может возникнуть, если запрос ввода/вывода содержит большой объем данных, если ограничены возможности передачи данных устройства или если испытывается недостаток системных ресурсов, что накладывает ограничения на размер буфера или число регистров отображения.

Для первой передачи запроса драйвер программирует устройство осуществлять передачу данных с самого начала буфера. Последующие же передачи транзакции должны начинаться с некоторым смещением относительно начала буфера. Чтобы определить смещение (и путем выведения узнать, является ли данная передача первой), драйвер вызывает метод *WdfDmaTransactionGetBytesTransferred*, передавая ему в параметрах дескриптор текущего объекта транзакции DMA. Метод возвращает число байтов уже переданных в транзакции или ноль, если еще не выполнилась ни одна передача. Драйвер может использовать эти возвращенные значения в качестве смещения буфера.

Определив смещение, драйвер должен организовать структуры данных, необходимые для программирования устройства. Специфичные подробности данного шага будут разными для различных устройств, но для типичной передачи DMA требуется базовый адрес и размер каждого компонента передачи. Функция обратного вызова *EvtProgramDma* получает пары "базовый адрес/длина" в параметре списка "разбиение/объединение". Число элементов в списке "разбиение/объединение" зависит от типа исполняемого DMA и типа устройства. Для пакетного DMA список содержит только одну пару. Для устройств с аппаратной поддержкой механизма "разбиение/объединение" список содержит несколько пар "адрес/размер". Драйвер преобразует эти пары в формат, воспринимаемый устройством.

После этого драйвер программирует устройство и начинает передачу. Прежде чем обращаться к регистрам устройства, драйвер захватывает спин-блокировку прерывания для устройства. Эта блокировка повышает уровень IRQL устройства до DIRQL, таким образом обеспечивая, что устройство не будет пытаться выдавать прерывания в то время, когда драйвер изменяет значения регистров. Так как код, защищенный этой блокировкой, исполняется на высоком уровне IRQL, драйвер обязан удерживать блокировку лишь в течение минимального периода времени. Блокировка должна защищать только код, который физически обращается к регистрам устройства. Все вычисления и отладка должны происходить вне пределов этой блокировки.

Если драйвер успешно начинает передачу DMA, то функция обратного вызова *EvtProgramDma* возвращает значение TRUE. В случае ошибки драйвер должен отменить текущую транзакцию и возвратить FALSE.

## Пример: обработка запроса

В листинге 17.3 приведена функция обратного вызова *EvtProgramDma* образца драйвера PLx5x9x из файла Read.c для обработки запросов на чтение. Большая часть кода этой функции специфична для устройства и опущена в данном листинге.

### Листинг 17.3. Функция обратного вызова *EvtProgramDma* образца драйвера PLx5x9x

```
BOOLEAN PLxEvtProgramReadDma (
    IN WDFDMA TRANSACTION Transaction,
    IN WDFDEVICE Device,
    IN WDFCONTEXT Context,
    IN WDF_DMA_DIRECTION Direction,
    IN PSCATTER_GATHER_LIST SgList)
{
    PDEVICE_EXTENSION devExt;
    size_t offset;
    PDMA_TRANSFER_ELEMENT dteVA;
    ULONG_PTR dteLA;
    BOOLEAN errors;
    ULONG i;

    devExt = PLxGetDeviceContext(Device);
    errors = FALSE;
    // Получаем число байтов, уже переданных в этой транзакции.
    offset = WdfDmaTransactionGetBytesTransferred(Transaction);
    // Организуем адреса для программирования устройства.
    ... // Код, специфичный для устройства, опущен.
    // Получаем спин-блокировку прерывания для устройства
    // и начинаем DMA.
    WdfInterruptAcquireLock( devExt->Interrupt );
    ... // Специфичный для устройства код, программирующий
        // регистры устройства для DMA.
    WdfInterruptReleaseLock( devExt->Interrupt );
    // В случае ошибки в функции обратного вызова EvtProgramDma
    // освобождаем объект транзакции DMA и завершаем запрос.
    if (errors) {
        NTSTATUS status;
        WDFREQUEST request;
        (VOID) WdfDmaTransactionDmaCompletedFinal(Transaction,
                                                    offset, &status);
        // Получаем соответствующий запрос из транзакции.
        request = WdfDmaTransactionGetRequest(Transaction);
        WdfDmaTransactionRelease(Transaction);
        WdfRequestCompleteWithInformation(request,
                                          STATUS_INVALID_DEVICE_STATE, 0);
        return FALSE;
    }
    return TRUE;
}
```

Первым делом драйвер инициализирует свои локальные переменные, после чего вызывает метод *WdfDmaTransactionGetBytesTransferred*, чтобы определить, сколько байтов данных уже было передано для этой транзакции. Возвращенное методом значение драйвер использует, чтобы определить смещение в буфере, с которого нужно начинать передачу.

Дальше драйвер преобразует пары "адрес/размер" в списке "разбиение/объединение" в формат, воспринимаемый устройством, и организует структуры данных, необходимые ему для программирования устройства. После этого драйвер вызывает метод `WdfInterruptAcquireLock`, чтобы получить спин-блокировку прерывания для устройства, обращается к регистрам устройства, чтобы запрограммировать устройство, и вызывает метод `WdfInterruptReleaseLock`, чтобы освободить спин-блокировку.

Если во время программирования устройства происходит ошибка, драйвер вызывает метод `WdfDmaTransactionDmaCompletedFinal`, который извещает инфраструктуру о том, что транзакция была завершена, но данные не были переданы полностью. В качестве параметров методу передаются: дескриптор объекта транзакции, число успешно переданных байтов и указатель, куда возвратить значение статуса. Поскольку метод определен булевого типа, он всегда возвращает значение `TRUE`, поэтому в образце драйвера его тип преобразовывается в `VOID`. Потом драйвер освобождает объект транзакции DMA для будущего использования, завершает запрос ввода/вывода со статусом неудачи `STATUS_INVALID_DEVICE_STATE` и возвращает `FALSE` из функции обратного вызова.

В случае успешного выполнения программирования устройства функция обратного вызова возвращает значение `TRUE`. Чтобы сообщить о завершении передачи, устройство выдает прерывание.

## Обработка завершения DMA

Обычно устройство сообщает о завершении передачи, выдавая прерывание. Драйвер выполняет минимальную обработку в функции обратного вызова `EvtInterruptIsr` и ставит в очередь функцию обратного вызова `EvtInterruptDpc`, которая обычно является ответственной за обработку завершенной передачи DMA.

### Завершение передачи, транзакции и запроса

Когда устройство сообщает о завершении передачи DMA, драйвер должен определить, была ли завершена вся транзакция DMA. Если вся транзакция была завершена, то драйвер завершает запрос ввода/вывода и удаляет или освобождает объект транзакции DMA.

Инфраструктура организует индивидуальные передачи DMA и отслеживает число байтов в каждой передаче. По завершению передачи драйвер извещает об этом инфраструктуру, вызывая метод `WdfDmaTransactionDmaCompleted` или `WdfDmaTransactionDmaCompleted-WithLength`. Обоим методам драйвер передает объект транзакции DMA и получает от них значение `NTSTATUS`. Единственная разница между этими двумя методами состоит в том, что метод `WdfDmaTransactionDmaCompletedWithLength` также принимает входной параметр, который указывает число байтов, переданных устройством в только что завершенной операции, что может быть полезным для устройств, которые возвращают эту информацию.

Оба эти метода делают следующее:

- ◆ сбрасывают все данные, оставшиеся в кэше Windows;
- ◆ освобождают разделяемые ресурсы, такие как регистры отображения, выделенные для поддержки передачи;
- ◆ определяют, если завершение передачи также завершает всю транзакцию DMA.

При положительном результате этой проверки методы возвращают `TRUE`. В противном случае возвращается `FALSE` и статус `STATUS_MORE_PROCESSING_REQUIRED`.

Если завершена все транзакция, то драйвер завершает соответствующий запрос ввода/вывода.

Если вся транзакция DMA еще не завершена, то для ее завершения потребуется еще одна или несколько дополнительных передач. Тогда инфраструктура выделяет необходимые для следующей передачи ресурсы и снова вызывает функцию обратного вызова *EvtProgramDma* драйвера для выполнения следующей передачи.

## Пример: обработка завершения DMA

В листинге 17.4, на примере драйвера PLX9x5x, демонстрируются шаги, выполняемые типичным драйвером KMDF для завершения передачи DMA. Данный код основан на обработке завершения операции чтения в функции обратного вызова *EvtInterruptDpc* драйвера в файле *Isrdpc.c*.

### Листинг 17.4. Обработка завершения DMA

```
if (readComplete) {
    BOOLEAN transactionComplete;
    WDFDMA TRANSACTION dmaTransaction;
    size_t bytesTransferred;
    // Получаем текущую транзакцию чтения DMA.
    dmaTransaction = devExt->CurrentReadDmaTransaction;
    // Извещаем о завершении данной операции DMA:
    // если имеются еще данные передачи, это может начать
    // передачу следующего пакета.
    transactionComplete =
        WdfDmaTransactionDmaCompleted(dmaTransaction, &status);
    if (transactionComplete) {
        // Завершаем транзакцию DMA и запрос.
        devExt->CurrentReadDmaTransaction = NULL;
        bytesTransferred = ((NT_SUCCESS(status)) ?
            WdfDmaTransactionGetBytesTransferred(dmaTransaction) : 0);
        WdfDmaTransactionRelease(dmaTransaction);
        WdfRequestCompleteWithInformation(request, status, bytesTransferred);
    }
}
```

Код в листинге 17.4 исполняется после выдачи устройством прерывания, извещающего о завершении операции чтения.

Исполнение в примере начинается с получения дескриптора объекта транзакции DMA для текущей операции чтения, который драйвер использует в вызове метода *WdfDmaTransactionDmaCompleted*. Этот метод извещает инфраструктуру о завершении текущей передачи транзакции DMA. Он возвращает булево значение, указывающее, была ли также завершена вся транзакция, и значение *NTSTATUS*, указывающее успех или неудачу операции.

Если метод *WdfDmaTransactionDmaCompleted* возвращает *TRUE*, драйвер завершает текущий запрос, присваивая нулевое значение переменной в области контекста его устройства, в которой хранится дескриптор текущего объекта DMA. В случае успешного завершения передачи драйвер получает число байтов, переданных в транзакции DMA, вызывая метод *WdfDmaTransactionGetBytes-Transferred*. Так как сейчас завершена вся транзакция DMA, драйвер освобождает объект транзакции DMA, вызывая для этого метод

`WdfDmaTransactionRelease`. Наконец, драйвер завершает запрос ввода/вывода обычным способом, вызывая метод `WdfRequestCompleteWithInformation` и передавая ему статус и число переданных байтов.

Если метод `WdfDmaTransactionDmaCompleted` возвращает `FALSE`, то функция обратного вызова `EvtInterruptDpc` больше не выполняет никаких операций для данной транзакции DMA, т. к. инфраструктура немедленно вызывает функцию обратного вызова `EvtProgramDma` драйвера для обработки следующей передачи транзакции.

## Тестирование драйверов DMA

Помощь в тестировании драйверов, работающих с DMA, могут оказать следующие три средства: инструмент Driver Verifier, расширение отладчика `!dma` и специфичные для KMDF расширения отладчика для DMA.

Дополнительную информацию о Driver Verifier см. в главе 21. Расширения отладчика обсуждаются в главе 22.

### Верификация, специфичная для DMA

В Windows XP и более поздних версиях Windows инструмент Driver Verifier содержит специальные тесты для выявления неправильного использования различных операций DMA. Инструмент Driver Verifier предоставляет средства для выявления следующих специфичных для DMA ошибок, которые связаны с базовыми структурами WDM, а не с объектами WDF:

- ◆ переполнение или опустошение буфера памяти DMA. Эти ошибки могут быть вызваны аппаратным обеспечением или драйвером;
- ◆ освобождение несколько раз одного и того же буфера, канала адаптера, регистра отображения или списка "разбиение/объединение";
- ◆ утечка памяти, в результате неосвобождения общих буферов, каналов адаптера, регистров отображения, списков "разбиение/объединение" или адаптеров;
- ◆ попытка использовать адаптер, который был освобожден и больше не существует;
- ◆ невыполнение сброса буфера адаптера;
- ◆ исполнение DMA на страничном буфере;
- ◆ одновременное выделение слишком большого числа регистров отображения или выделение больше регистров отображения, чем максимально позволенное число;
- ◆ попытка освободить регистры отображения в то время, когда некоторые из них еще используются;
- ◆ попытка сбросить регистр отображения, который не был задействован;
- ◆ вызов процедур DMA на неправильном уровне IRQL.

В добавление ко всему только что перечисленному, инструмент Driver Verifier предоставляет средства для выполнения нескольких второстепенных проверок на непротиворечивость, которые описываются в разделе **DMA Verification** (Верификация DMA) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80070>.

Многие из этих ошибок возникают при операциях, выполняемых инфраструктурой, а не драйвером KMDF. Но некоторые из них могут также служить признаком проблем в коде драйвера.

Инструмент Driver Verifier предоставляет еще одну возможность для облегчения тестирования драйверов для устройств DMA. При исполнении Driver Verifier выполняет двойную буферизацию всех передач DMA для проверяемых драйверов. Двойная буферизация помогает обеспечить использование драйвером правильных адресов для операций DMA.

В зависимости от типа обнаруженной им ошибки Driver Verifier извещает о специфичных для DMA ошибках либо выдавая ASSERT, либо генерируя останов bugcheck с кодом 0x6E (т. е. DRIVER\_VERIFIER\_DMA\_VIOLATION).

При тестировании драйверов с помощью Driver Verifier не следует забывать, что это не автоматический инструмент. Его способности ограничены наблюдением за операциями, исполняемыми драйвером, и выдачей предупреждения при обнаружении неправильного исполнения операций драйвером. Поэтому необходимо выполнить всестороннюю проверку работы драйвера с многообразными запросами ввода/вывода при включенном Driver Verifier, в том числе запросы на чтение и запись разной длины.

## Расширение отладчика *!dma*

Еще одним средством отладки драйверов устройств DMA является расширение отладчика *!dma*. Это расширение выводит на экран информацию о подсистеме DMA и драйверах устройств DMA, проверяемых Driver Verifier.

Если верификация DMA для драйвера не включена, то расширение отладчика *!dma* может создать список всех адаптеров DMA в системе. Объект адаптера DMA является представлением WDM возможностей DMA устройства. Инфраструктурный объект выключателя DMA представляет один или несколько нижележащих адаптеров DMA устройства. При включенной верификации DMA расширение отладчика *!dma* может также перечислить следующую информацию:

- ◆ объект устройства, регистры отображения, списки "разбиение/объединение" и общие буферы, ассоциированные с каждым адаптером DMA;
- ◆ использование регистров отображения для определенного адаптера DMA, включая информацию о том, выполняется ли двойная буферизация передач устройства;
- ◆ размер и виртуальные и физические адреса любых сегментов общего буфера.

Эта информация относится к нижележащим в стеке структурам данных инфраструктуры, а не к объектам, создаваемым собственно драйвером KMDF. Тем не менее она может быть полезной при отладке и помочь выяснить, работает ли DMA так, как ожидалось.

Подробное описание расширения отладчика *!dma* приводится в документации, сопровождающей пакет Debugging Tools for Windows.

## Расширения отладчика KMDF для DMA

Расширения отладчика KMDF, поставляющиеся с WDK, включают несколько команд, предназначенные специально для отладки драйверов устройства DMA. Эти расширения и их краткое описание перечислены в табл. 17.3.

Дополнительную информацию о расширениях отладчика для KMDF см. в главе 22.

В листинге 17.5 приведен пример вывода, сделанного расширениями отладчика DMA.

**Таблица 17.3. Расширения отладчика KMDF для верификации DMA**

<b>Расширения отладчика</b>	<b>Соответствующий объект</b>	<b>Исполняемые операции</b>
!wdfdmaenablers	WDFDEVICE	Составляет список всех выключателей DMA и их транзакций и объектов общего буфера
!wdfdmaenabler	WDFDMAENABLER	Выводит информацию об определенном объекте выключателя DMA и его транзакциях и объектах общего буфера
!wdftransaction	WDFDMATRANSACTION	Выводит информацию о данном объекте транзакции
!wdfcommonbuffer	WDFCOMMONBUFFER	Выводит информацию о данном объекте общего буфера

**Листинг 17.5. Вывод, сделанный расширениями отладчика KMDF для DMA**

```
0: kd> !wdfdmaenablers 0x7e6128a8
Dumping WDFDMAENABLERS 0x7e6128a8
=====
1) !WDFDMAENABLER 0x7e7ac630:
-----
List of Transaction objects:
1) !WDFDMATRANSACTION 0x016f5fb0
List of CommonBuffer objects:
1) IWDFCOMMONBUFFER 0x7e887c00
2) IWDFCOMMONBUFFER 0x7e780cd0
3) IWDFCOMMONBUFFER 0x7ec6b338
0: kd> !WDFDMAENABLER 0x7e7ac630
Dumping WDFDMAENABLER 0x7e7ac630
=====
Profile: WdfDmaProfileScatterCather64Duplex
Read DMA Description:
    WDM AdapterObject (!dma): 0x81b37938
    Maximum fragment length: 32768
    Number of map registers: 9
Write AdapterObject Description:
    WDM AdapterObject (!dma): 0xff0b92d0
    Maximum fragment length: 32768
    Number of map registers: 9
Default common buffer alignment: 1
Max SC-elements per transaction: 0xffffffff
List of Transaction objects:
1) !WDFDMATRANSACTION 0x016f5fb0
    State: FxDmaTransactionStateTransfer
    Direction: Read
    Associated WDFREQUEST: 0x0108b2d0
    Input Mdl: 0x81b042b8
    Starting Addr: 0x81893000
    Mdl of current fragment being transferred: 0x81b042b8
    VA of current fragment being transferred: 0x81893000
    Length of fragment being transferred: 0x8c
    Max length of fragment: 0x8000
```

```
Bytes transferred: 0x0
Bytes remaining to be transferred: 0x0
ProgramDmaFunction: (0xf7cc62e0) testdma!EvtProgramDma
List of CommonBuffer objects:
1) !WDFCOMMONBUFFER 0x7e887c00
   Virtual address allocated: 0xffff8ca560 Aligned: 0xffff8ca560
   Logical address allocated: 0x18b79560 Aligned: 0x18b79560
   Length allocated: 0x1a Length requested: 0xa
   Alignment Mask: 0x10
```

## Рекомендации к разработке драйверов DMA

Далее приводятся несколько подсказок и полезных советов для написания драйверов DMA для Windows.

- ◆ Страйтесь получить как можно лучшее понимание абстракции DMA Windows. Понимание абстракции поможет вам в написании и отладке ваших драйверов DMA.
  - ◆ Не применяйте методов в обход поддержки DMA, предоставляемой KMDF, или в обход абстракции DMA Windows, в попытке создать собственное решение.
- Например, никогда не является приемлемым создание содержимого списка "разбиение/объединение" вручную, путем интерпретирования содержимого списка MDL, и использование полученных данных для программирования устройства DMA. При таком подходе не принимается во внимание возможное применение регистров отображения.
- ◆ Драйверы, которые получают запросы DMA из очереди с последовательной диспетчериацией, что обеспечивает только одну активную транзакцию в любое время, должны писаться таким образом, чтобы во время инициализации DMA они создавали только один объект транзакции DMA и использовали этот объект повторно для последующих запросов DMA.
  - ◆ Выполните проверку разрабатываемых драйверов общими средствами Driver Verifier в целом и средствами верификации DMA в частности.

Кроме разработки драйвера для правильного использования поддержки DMA, встроенной в KMDF, всестороннее тестирование с применением Driver Verifier является, возможно, единствено наиболее важным, что можно сделать, чтобы обеспечить правильную работу драйвера под Windows.

# ГЛАВА 18

## Введение в СОМ

Для создания драйвера UMDF необходимо использовать несколько объектов СОМ (Component Object Model, модель составных объектов), которые принадлежат к среде исполнения UMDF; кроме этого, необходимо создать несколько объектов обратного вызова на основе технологии СОМ. Хотя приложения на основе технологии СОМ снискали репутацию больших, сложных и трудных в реализации, большая часть этих недостатков обусловлена средой исполнения СОМ и требованиями приложений и не является присущей технологии СОМ как таковой.

В UMDF применяется только наиболее необходимые компоненты модели программирования СОМ, в которые не входит среда исполнения СОМ, что делает драйверы UMDF легковесными и легко реализуемыми. В этой главе дается введение в основы использования и создания СОМ объектов для нужд UMDF. В особенности она направлена на разработчиков драйверов режима ядра, имеющих ограниченный опыт работы с СОМ.

Ресурсы, необходимые для данной главы	Местонахождение
<b>Инструменты и файлы</b>	
Wudfddi.idl	%wdk%\BuildNumber\inc\wdflumdf\VersionNumber
<b>Справочные материалы</b>	
Раздел "Component Object Model" в MSDN	<a href="http://go.microsoft.com/fwlink/?LinkId=79770">http://go.microsoft.com/fwlink/?LinkId=79770</a>
Книга "Inside COM" <sup>1</sup> (издательство Microsoft Press)	<a href="http://go.microsoft.com/fwlink/?LinkId=79771">http://go.microsoft.com/fwlink/?LinkId=79771</a>

## Прежде чем приступить

Объекты СОМ в общем и драйверы UMDF в частности почти всегда пишутся на языке C++. Для целей этой главы предполагается, что вы обладаете надлежащими знаниями в области объектно-ориентированного программирования (ООП) в общем и программирования на языке C++ в частности. Вы должны понимать такие основные концепции, как:

- ◆ структура класса: ключевые слова `struct` и `class`, открытые (`public`) и закрытые (`private`) члены, статические методы, конструкторы, деструкторы и чисто абстрактные классы;

---

<sup>1</sup> Основы СОМ. — Пер.

- ◆ создание и уничтожение объектов с помощью операторов new и delete соответственно;
- ◆ наследование, включая базовые и производные классы, множественное наследование и чисто виртуальные методы.

Если вы новичок в программировании на языке C++, то вам следует хорошо ознакомиться с этими основными понятиями. Другие аспекты языка C++, такие как перегрузка операторов или шаблоны, не являются необходимыми для драйверов UMDF. В драйверах UMDF можно использовать стандартные библиотеки шаблонов языка C++, такие как Standard Template Library (STL) или Active Template Library (ATL). Но использование этих библиотек не является обязательным и они не применяются в примерах в этой главе.

### Полезная информация

Объекты COM технически возможно, хотя значительно менее удобно, реализовать с помощью языка С. Но такой подход применяется сравнительно редко и здесь не рассматривается.

## Структура драйвера UMDF

Для начала будет полезным понять, каким образом драйверы UMDF используют COM. Драйверы UMDF реализуются в виде внутрипроцессных (in-process) объектов COM, которые иногда сокращенно называются InProc-объектами или серверами COM. Внутрипроцессные объекты упаковываются в DLL и исполняются в контексте потока их хоста. В DLL может содержаться любое число внутрипроцессных объектов COM.

Все DLL UMDF содержат, по крайней мере, три обязательных сервера COM и обычно несколько необязательных серверов, точное число которых зависит от требований устройства. DLL также должна экспортировать две стандартные функции. Компоненты типичной DLL драйвера UMDF показаны на рис. 18.1.

Далее приводится краткое описание основных компонентов DLL, показанных на рис. 18.1. Более подробно эти компоненты рассматриваются далее в этой главе.

- ◆ Функции DllMain и DllGetClassObject.

Эти две стандартные функции предоставляют основную структуру DLL и обычно являются единственными функциями, которые экспортятся по имени.

- ◆ Фабрика классов для объекта обратного вызова драйвера.

Фабрика классов представляет собой специальный объект COM, который клиент может использовать для создания экземпляра определенного объекта COM.

Среда исполнения UMDF начинает загружать драйвер, используя фабрику классов для создания объекта обратного вызова драйвера, который является единственным объектом обратного вызова, для которого требуется фабрика классов.

- ◆ Объекты обратного вызова UMDF.

Драйверы UMDF в основном состоят из набора объектов обратного вызова COM, используемых средой исполнения UMDF для взаимодействия с драйвером.

Два объекта обратного вызова, показанные на рис. 18.1, являются обязательными для всех драйверов. Наличие любых других объектов обратного вызова не является обязательным и зависит от требований конкретного драйвера.

◆ Интерфейсы.

Объекты COM не предоставляют индивидуальных методов. Вместо этого, объекты COM группируют методы в интерфейсы и предоставляют интерфейсы. Все объекты обратного вызова UMDF предоставляют, по крайней мере, два, а иногда и больше интерфейсов.

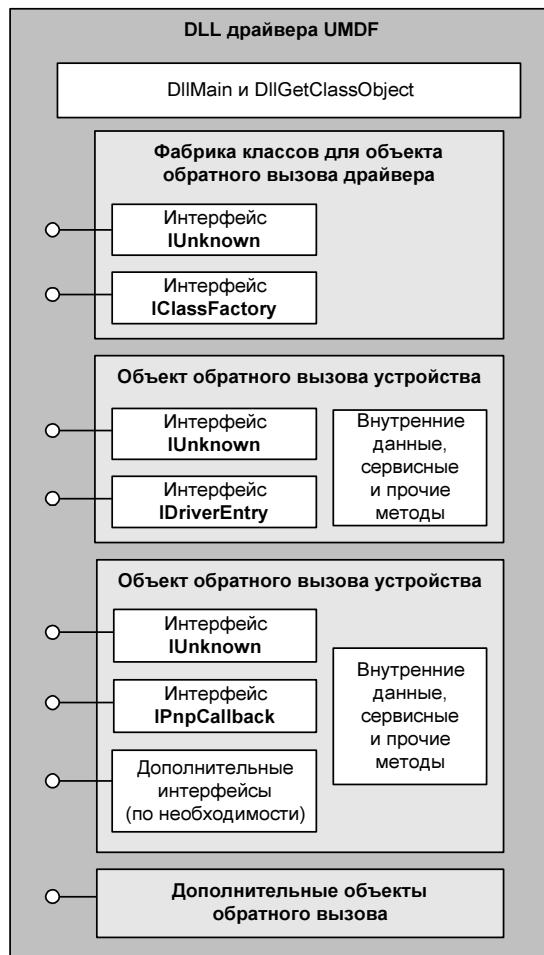


Рис. 18.1. Схема DLL драйвера UMDF

Среда исполнения UMDF также содержит несколько объектов COM, которые драйверы должны использовать для взаимодействия со средой исполнения. Процесс, использующий объект COM, называется *клиентом COM*. Таким образом, реализация драйвера в основном состоит из реализации нескольких объектов обратного вызова COM, которые работают в качестве клиентов COM для взаимодействия со средой исполнения UMDF. В этой главе рассматриваются основы применения COM для реализации компонентов DLL, показанных на рис. 18.1.

## Краткий обзор СОМ

Объект СОМ инкапсулирует данные и предоставляет открытые методы, с помощью которых приложения могут давать указания объектам выполнять различные задачи. Объекты СОМ похожи на объекты, используемые в других моделях ООП, особенно в языке C++. Но между СОМ и обычным программированием в C++ существует несколько важных различий.

### ◆ Независимость от языка программирования.

Технология СОМ не основана ни на каком конкретном языке программирования; это двоичный стандарт. Теоретически объекты СОМ можно реализовать на любом языке программирования, включая язык С. Но для практических целей язык C++ предоставляет наиболее прямолинейный способ реализации объектов СОМ и является языком, обычно применяемым для написания драйверов UMDF.

### ◆ Реализация объектов и взаимодействие между классами.

Объекты СОМ обычно реализуются в виде классов C++, но структура класса объекта является невидимой для клиентов объекта. На практике, для использования объекта СОМ не требуется ничего знать о том, каким образом реализована его основа. Все, что необходимо знать, — это, какие интерфейсы предоставляет объект и как эти интерфейсы использовать.

Взаимосвязь между объектом в его открыто предоставленном виде и лежащим в его основе классом или классами C++ не обязательно является простой. Как разработчик, вы определяете внутренние подробности реализации. Но хорошим принципом разработки СОМ будет, чтобы класс представлял логическую единицу.

### ◆ Время жизни объекта.

Клиенты СОМ управляют временем жизни создаваемых ими объектов СОМ с помощью специфичных для СОМ методов, а не через операторы `new` и `delete` языка C++.

### ◆ Наследование.

СОМ не поддерживает обычное наследование ООП. Наследование широко применяется для реализации объектов СОМ, но эта подробность реализации не предоставляется внешне.

### ◆ Инкапсуляция.

Инкапсуляция объектов СОМ осуществляется более строго, чем обычных объектов C++. Клиенты СОМ не используют "сырые" указатели на объекты для доступа к объектам СОМ. В СОМ нельзя просто создать экземпляр объекта СОМ и вызвать любой открытый метод этого объекта. Вместо этого, открытые методы СОМ группируются в интерфейсы. Чтобы использовать методы в данном интерфейсе, необходимо получить указатель интерфейса. В некоторых случаях неизвестно, или нет необходимости знать, какой объект предоставляет определенный интерфейс.

### ◆ Интерфейсы и объекты.

Все интерфейсы СОМ происходят от основного СОМ-интерфейса `IUnknown`. Этот интерфейс предоставляется всеми объектами СОМ и является необходимым для работы объекта.

Указатель интерфейса позволяет клиенту СОМ использовать любые методы этого интерфейса. Но с его помощью нельзя получить доступ к методам никаких других интерфейсов, которые могут предоставляться объектом. Для получения указателя на другие интерфейсы объекта необходимо применить метод `IUnknown::QueryInterface`.

◆ Элементы данных.

Объекты COM не могут предоставлять открытых элементов данных; вместо этого они предоставляют данные посредством методов, называющихся *методами доступа* (accessors). Методы доступа являются обычными методами, но они отличаются от методов, применяемых для выполнения задач, особым форматом именования. Как правило, для чтения и записи применяются отдельные методы доступа.

## Содержимое объекта COM

На рис. 18.2 показаны взаимоотношения между типичным объектом COM UMDF (объектом обратного вызова устройства UMDF) и его содержимым. В оставшейся части этого раздела рассматриваются базовые компоненты COM и их работа.

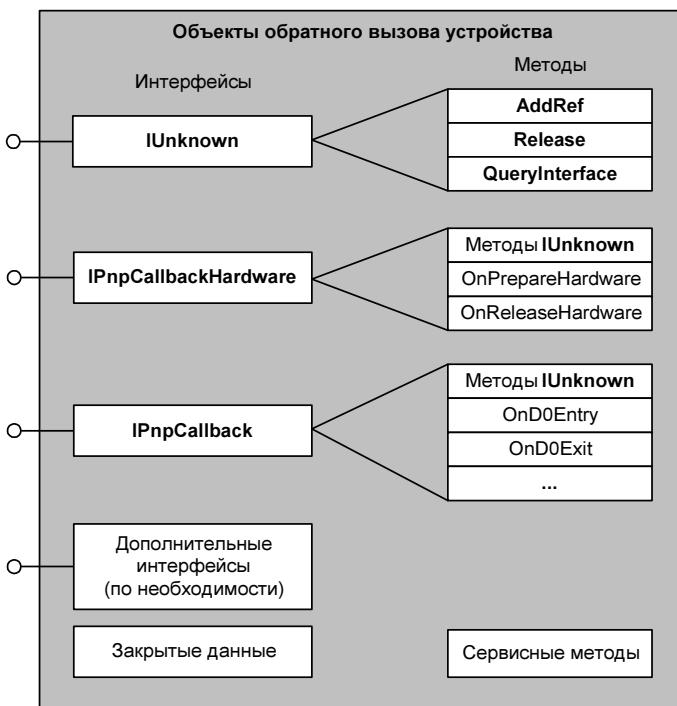


Рис. 18.2. Типичный объект COM

## Объекты и интерфейсы

Типичный объект COM предоставляет, по крайней мере, два, а иногда и больше интерфейсов. Интерфейс — это механизм COM для предоставления открытых методов объекта. В традиционных языках ООП, таких как C++, клиенты используют "сырые" указатели объектов, которые предоставляют доступ ко всем открытым методам и элементам данных, поддерживаемых объектом. В COM же объекты группируют методы в интерфейсы, которые они потом и предоставляют. Чтобы использовать методы в данном интерфейсе, клиент сначала должен получить указатель этого интерфейса. Объекты COM никогда не предоставляют элементы данных прямым образом.

В сущности, интерфейс представляет собой объявление, в котором указывается группа методов, их синтаксис и их общая функциональность. Любой объект COM, для которого требуется функциональность интерфейса, может реализовать методы и предоставить этот интерфейс. Некоторые интерфейсы высоко специализированы и предоставляются только одним объектом, в то время как другие интерфейсы представляются различными типами объектов. Например, все объекты COM должны предоставлять интерфейс `IUnknown`, который используется для управления объектом.

Традиционно, имена интерфейсов всегда начинаются с заглавной буквы "I". Имена интерфейсов, предоставляемых объектами UMDF, начинаются с буквосочетания `IWDF`. Общепринятый метод представления метода в документации заимствован из синтаксиса для объявлений языка C++. Например, метод `CreateRequest` интерфейса `IWDFDevice` представляется в виде `IWDFDevice::CreateRequest`.

Но если ошибочное толкование маловероятно, то имя интерфейса часто опускается.

В COM нет общепринятых стандартов для именования объектов. Так как при программировании в COM основное внимание уделяется интерфейсам, а не лежащим в их основе объектам, то имена объектов имеют второстепенное значение.

Если объект предоставляет стандартный интерфейс, он должен реализовывать все методы этого интерфейса. Но подробности реализации этих методов могут быть разными для разных объектов. Например, для реализации определенного метода разные объекты могут использовать разные алгоритмы. Единственным жестким требованием является использование в методах правильного синтаксиса.

После публикации открытого интерфейса (который включает все интерфейсы UMDF) объявление интерфейса изменять нельзя. Фактически интерфейс является контрактом между объектом и любым клиентом, которой возможно воспользуется им. Если изменить объявление интерфейса, то попытка клиента использовать интерфейс на основании его предыдущего объявления завершится неудачей.

## Интерфейс `IUnknown`

Интерфейс `IUnknown` является ключевым интерфейсом COM. Все объекты COM должны предоставлять этот интерфейс и все реализуемые интерфейсы должны наследовать от него. Интерфейс `IUnknown` имеет два основных назначения: он предоставляет доступ к интерфейсам объекта и управляет подсчетом ссылок, таким образом контролируя время жизни объекта. Эти задачи выполняются следующими тремя методами:

- ◆ метод `QueryInterface` вызывается, чтобы запросить указатель на один из интерфейсов объекта;
- ◆ метод `AddRef` вызывается, чтобы увеличить значение счетчика ссылок при создании нового указателя на интерфейс;
- ◆ метод `Release` вызывается, чтобы уменьшить значение счетчика ссылок при освобождении указателя на интерфейс.

Указатель на интерфейс `IUnknown` часто используется в качестве функционального эквивалента указателя объектов, т. к. все объекты предоставляют интерфейс `IUnknown`, который предоставляет доступ к другим интерфейсам объекта. Но указатель на интерфейс `IUnknown` не обязательно такой же, как и указатель на объект, т. к. он может быть смешен по отношению к базовому адресу объекта.

## Подсчет ссылок

В отличие от объектов C++, время жизни объектов COM непосредственно не управляется их клиентами. Вместо этого, объект COM ведет учет ссылок следующим образом.

- ◆ Новый созданный объект имеет один активный интерфейс, и значение его счетчика ссылок равно единице.
- ◆ При каждом запросе клиентом другого интерфейса на объекте значение счетчика ссылок увеличивается.
- ◆ По окончанию работы клиента с интерфейсом, он освобождает указатель на интерфейс и уменьшает на единицу значение счетчика ссылок.
  - Клиенты обычно увеличивают или уменьшают счетчик ссылок, когда, например, они создают или уничтожают копию указателя интерфейса.
  - По освобождению всех указателей интерфейсов объекта, значение счетчика ссылок становится нулевым, и объект самоуничтожается.

В следующем разделе приводится несколько рекомендаций по использованию методов `AddRef` и `Release` для правильного управления счетчиком ссылок объекта.

### Внимание!

Необходимо быть чрезвычайно осторожным со счетчиком ссылок при работе или реализации объектов COM. Хотя клиенты не уничтожают объекты COM явным образом, в COM нет механизма сборки мусора для обработки неиспользуемых объектов или объектов вне области видимости, как это делается в управляемом коде. Одной из распространенных ошибок является упщение освободить интерфейс. В таком случае значение счетчика ссылок никогда не станет нулевым, и объект будет находиться в памяти бесконечно. С другой стороны, освобождение указателя интерфейса слишком много раз уничтожает объект преждевременно, что может вызвать фатальный сбой драйвера. Еще хуже, если обнаружение ошибок, вызванных неправильным подсчетом ссылок, может быть очень трудной задачей.

## Рекомендации по использованию методов `AddRef` и `Release`

Для правильного управления счетчиком ссылок объекта придерживайтесь следующих общих рекомендаций по использованию методов `AddRef` и `Release`.

### Подсчет ссылок

Обычно клиентам COM не требуется вызывать метод `AddRef` явно. Вместо этого метод, возвращающий указатель интерфейса, уменьшает значение счетчика ссылок. Основным исключением из этого правила является копирование указателя интерфейса. В таком случае, чтобы уменьшить значение счетчика ссылок объекта, нужно явно вызвать метод `AddRef`. По окончании работы с копией указателя вызывается метод `Release`.

### Использование указателей интерфейса в качестве параметров

Правильный подход при передаче указателя интерфейса методу в качестве параметра зависит от типа параметра и от того, использует или реализует драйвер данный метод.

- ◆ Входной (IN) параметр.

*Метод используется:* если вызывающий клиент передает указатель интерфейса в качестве входного параметра, то он и является ответственным за освобождение указателя.

*Метод реализуется:* если вызывающий клиент получает указатель интерфейса в качестве входного параметра, то он не должен освобождать указатель. Указатель не освобождается до тех пор, пока метод не возвратит управление, поэтому не существует риска, что значение счетчика ссылок станет нулевым, в то время, пока метод использует интерфейс.

◆ **Выходной (OUT) параметр.**

*Метод используется:* вызывающий клиент обычно передает указатель интерфейса со значением `NULL`, который указывает на действительный интерфейс по возвращению управления методом. Вызывающий клиент должен освободить этот указатель по окончании работы с ним.

*Метод реализуется:* если метод получает указатель интерфейса в качестве выходного параметра, значение указателя обычно установлено в `NULL`. Метод должен назначить этому параметру указатель на действительный интерфейс. Значение счетчика ссылок для интерфейса необходимо увеличить на единицу.

◆ **Входной/выходной (IN/OUT) параметр.**

Входные/выходные параметры для указателей интерфейса в UMDF не применяются.

## Вызовы метода `Release`

Для создания копии указателя интерфейса клиент COM обычно вызывает метод `AddRef`, а по окончании работы с указателем — вызывается метод `Release`. За исключением случаев, когда вы абсолютно уверены, что время жизни первоначального указателя превысит время жизни его копии, нельзя опускать вызовы этих методов. Рассмотрим примеры.

- ◆ Если для создания копии переданного ему указателя интерфейса метод А использует локальную переменную, вызов методов `AddRef` или `Release` не является обязательным. Когда функция возвращает управление, копия указателя выходит из области видимости, прежде чем вызывающий клиент имеет возможность освободить первоначальный указатель.
- ◆ Если для создания копии переданного ему указателя интерфейса метод А использует элемент данных или глобальную переменную, он должен вызвать метод `AddRef`, а по окончании работы с указателем — вызвать метод `Release`. В многопоточном приложении часто невозможно знать, могут ли другие методы также обращаться к той же самой глобальной переменной. Если метод А не вызывает метод `AddRef`, то другой метод может неожиданно уменьшить счет ссылок до нуля, уничтожая объект и делая недействительной копию указателя метода А. Но при условии, что метод А вызывает метод `AddRef`, когда он копирует указатель, объект не может быть уничтожен до тех пор, пока метод А не вызовет метод `Release`.

Если у вас имеются какие-либо сомнения, то следует вызывать метод `AddRef` при копировании указателя интерфейса и метод `Release` по окончании использования этой копии. Это окажет лишь незначительный отрицательный эффект на производительность, но будет намного безопасней.

## Исправление ошибок со счетчиком ссылок

Обнаружив в драйвере проблемы со счетчиком ссылок, не пытайтесь исправить их, простым добавлением вызовов методов `AddRef` или `Release`. Убедитесь в том, что драйвер получает и освобождает ссылки согласно правилам. В противном случае вы можете обнаружить, например, что вызов метода `Release`, добавленный в попытке исправить проблему с утечкой памяти, время от времени удаляет объект преждевременно и вызывает фатальный сбой.

## Идентификаторы GUID

Глобальные уникальные идентификаторы (GUID) широко применяются в программном обеспечении Windows. В COM идентификаторы GUID применяются для идентификации двух основных компонентов.

### ◆ Интерфейсов.

Идентификатор интерфейсов (IID) — это идентификатор GUID, который уникально идентифицирует определенный интерфейс COM. Независимо от того, какой объект предоставляет его, интерфейс всегда имеет один и тот же идентификатор IID.

### ◆ Классов.

Идентификатор классов (CLSID) — это идентификатор GUID, который уникально идентифицирует определенный объект COM. Идентификаторы CLSID требуются для объектов COM, создаваемых фабрикой класса, но являются необязательным для объектов, создаваемых другим образом. В UMDF фабрики класса и идентификаторы CLSID имеют только объекты обратного вызова драйвера.

Чтобы упростить использование идентификаторов GUID, в соответствующем заголовочном файле обычно определяются дружественные имена, которые традиционно состоят из префикса IID\_ или CLSID\_ и собственно описательного имени. Например, дружественное имя для идентификатора GUID, ассоциированного с интерфейсом `IDriverEntry`, будет `IID_IDriverEntry`. Для удобства, в документации UMDF интерфейсы обычно обозначаются именами, используемыми в их реализации, например `IDriverEntry`, а не идентификаторами IID.

## Таблица VTables

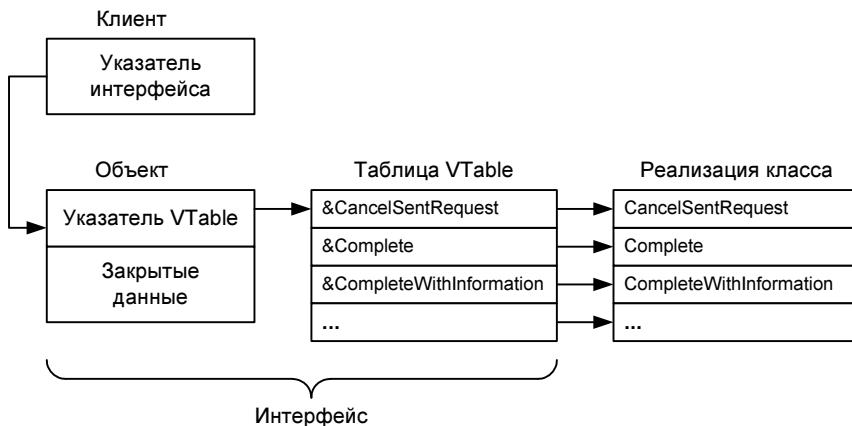
Доступ к объектам COM выполняется исключительно через таблицу виртуальных функций (virtual function table), которая часто называется сокращенно VTable. В этой таблице определяется структура физической памяти интерфейса. Таблица VTable представляет собой массив указателей на реализации всех методов, предоставляемых интерфейсом.

Когда клиент получает указатель на интерфейс, этот интерфейс обычно является указателем на указатель в таблице VTable, который в свою очередь указывает на таблицу указателей методов. Например, на рис. 18.3 показана структура памяти таблицы VTable для интерфейса `IWDFIoRequest`.

Таблица VTable представляет собой точно такую же структуру памяти, которую многие компиляторы языка C++ создают для чисто абстрактного базового класса. Это одна из главных причин, по которой объекты COM обычно реализовываются на языке C++ с объявлением интерфейсов как чисто абстрактных базовых классов. Интерфейсы объектов можно реализовать с помощью механизма наследования C++, и компилятор автоматически создаст таблицу VTable.

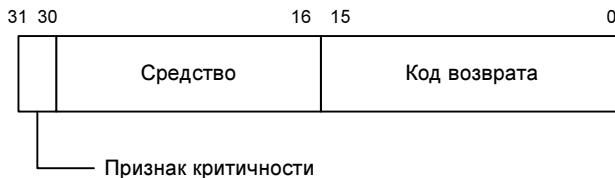
## Полезная информация

Взаимоотношение между чисто абстрактными базовыми классами и структурой таблицы VTable на рис. 18.3 не присуща C++, а является частью реализации компилятора. Но компиляторы для C++ компании Microsoft всегда создают правильную структуру таблицы VTable.

Рис. 18.3. Таблица VTable для интерфейса `IWDFIoRequest`

## HRESULT

Методы СОМ часто возвращают 32-битное значение, называющееся `HRESULT`. Это значение подобно значению `NTSTATUS`, которое возвращается процедурами драйвера режима ядра, и применяется по большому счету таким же образом. Структура значения `HRESULT` показана на рис. 18.4.

Рис. 18.4. Структура значения `HRESULT`

`HRESULT` разбито на три поля:

- ◆ **признак критичности (severity)** — в сущности, булево значение, указывающее успех или неудачу выполнения функции;
- ◆ **средство (facility)** — обычно можно игнорировать;
- ◆ **код возврата (return code)** — предоставляет подробное описание результатов.

Так же, как и в случае со значениями `NTSTATUS`, разбирать значение `HRESULT`, анализируя поля, в большинстве случаев не требуется. Стандартные значения `HRESULT` определены в заголовочных файлах и описываются в справочном материале для методов. По соглашению имена кодов успеха начинаются с `"S_"`, а неудачи — с `"E_"`. Например, код `S_OK` — это стандартное значение `HRESULT` для обычного успешного выполнения.

### Внимание!

Хотя `NTSTATUS` и `HRESULT` похожи друг на друга, они не взаимозаменяемы. Но иногда информацию в формате значения `NTSTATUS` необходимо возвратить как `HRESULT`. В таком слу-

чес используйте макрос `HRESULT_FROM_NT`, чтобы преобразовать значение `NTSTATUS` в эквивалентное значение `HRESULT`. Но этот макрос нельзя использовать для значения `NTSTATUS STATUS_SUCCESS`. Вместо этого нужно возвратить значение `HRESULT S_OK`. Если необходимо возвратить значение ошибки Windows, его можно преобразовать в `HRESULT` с помощью макрока `HRESULT_FROM_WIN32`.

Важно не рассматривать `HRESULT` как значения ошибок. Методы часто возвращают разные значения при успехе и неудаче. Например, обычно при успешном выполнении возвращается код `S_OK`, но иногда методы могут возвращать другие коды успешного выполнения, например, `S_FALSE`. Чтобы просто определить успешное или неуспешное выполнение метода, достаточно лишь значения признака критичности.

Чтобы не выполнять разборку `HRESULT` для получения значения признака критичности, в COM имеются два макрока, которые работают в значительной мере подобно макросу `NT_SUCCESS`, который применяется для проверки значений `NTSTATUS` на успех или неудачу. Для возвращенного значения `hr HRESULT`:

- ◆ макрос `FAILED(hr)` возвращает `TRUE`, если код критичности указывает неудачу, и `FALSE`, если — успех;
- ◆ макрос `SUCCEEDED(hr)` возвращает `FALSE`, если код критичности указывается неудачу, и `TRUE`, если — успех.

Возвращенный код `HRESULT` можно проанализировать, чтобы определить, требует ли ошибка принятия мер. Обычно возвращенное значение `HRESULT` просто сравнивается со списком возможных возвращаемых значений, приведенных в справочном материале для метода. Но имейте в виду, что эти списки часто не являются исчерпывающими. Обычно в них даются лишь значения `HRESULT`, специфичные для метода, или стандартные значения, имеющие специфичный для метода смысл. Метод же может возвращать и другие значения `HRESULT`.

Независимо от того, выполняете ли вы проверку на специфичные значения `HRESULT`, всегда осуществляйте проверку на простой успех или неудачу выполнения метода с помощью макросов `SUCCEEDED` или `FAILED`. В противном случае, если, например, вы выполняете проверку на успех, сравнивая возвращенное значение `HRESULT` с `S_OK`, а метод неожиданно возвратит значение `S_FALSE`, выполнение вашего кода, скорее всего, завершится неудачей.

## Свойства и события

Объекты COM предоставляют только методы; они не предоставляют ни свойств, ни событий. Но применяются другие механизмы, имеющие, по большому счету, то же самое назначение, что и эти средства.

◆ **Свойства.** Во многих моделях ООП свойства используются для предоставления элементов данных объекта управляемым и синтаксически простым способом. Объекты COM предоставляют функциональный эквивалент свойств, предоставляя элементы данных посредством методов доступа (accessor methods). Обычно, для читаемых и записываемых данных один метод доступа считывает значение данных, а второй — записывает. Так как методы доступа являются обычными методами, лишь служащими определенным целям, они отличаются от методов, применяемых для выполнения задач, особым форматом именования.

В UMDF для имен методов доступа для чтения и записи применяются префиксы `Get/Set` или `Retrieve/Assign` соответственно. Разница между этими двумя типами префиксовых заключается в том, что методы доступа типа `Get/Set` никогда не завершаются неудачей и не возвращают ошибок. А методы доступа типа `Assign/Retrieve` могут завершаться неудачей

и возвращают ошибки. Например, метод доступа `IWDFDevice::GetPnPState` получает состояние указанного свойства Plug and Play.

- ◆ **События.** Во многих моделях ООП события применяются для извещения объектов о происшествиях, требующих их внимания. Прежде чем один объект COM может известить другой объект COM о событии, между двумя объектами необходимо явно установить канал связи. Обычно для этого применяется метод, при котором объект-приемник события передает указатель интерфейса объекту-источнику события. После этого объект-источник может вызывать методы этого интерфейса для извещения объекта-приемника о произошедшем событии и, необязательно, передавать ему данные, связанные с событием.

Этот событийный механизм широко применяется в UMDF. Например, когда драйвер создает очередь для обработки запросов IOCTL устройства, он создает объект обратного вызова, который предоставляет интерфейс обратного вызова `IQueueCallbackDeviceIoControl`. После этого объект обратного вызова передает указатель своего интерфейса `IQueueCallbackDeviceIoControl` инфраструктурному объекту устройства. Когда среда исполнения UMDF получает запрос IOCTL устройства, инфраструктурный объект устройства вызывает метод `IQueueCallbackDeviceIoControl::OnDeviceIoControl`, чтобы известить драйвер о необходимости обработать запрос.

## Библиотека ATL

Библиотека ATL (Active Template Library, библиотека шаблонных классов) представляет собой набор шаблонных классов C++, которые часто применяются для упрощения процесса создания объектов COM. Разработчики драйверов UMDF могут воспользоваться библиотекой ATL для упрощения некоторых аспектов реализации драйверов. Но библиотека ATL не является необходимой для драйверов UMDF и не рассматривается в данной книге.

Информацию о библиотеке ATL см. в книге "Inside ATL" (издательство Microsoft Press) и документацию в библиотеке MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=79772>.

## Файлы IDL

Файлы IDL (Interface Definition Language, язык описания интерфейсов) предоставляют структурированный способ для спецификации интерфейсов и объектов. Файлы IDL пропускаются через компилятор IDL (компилятор, поставляемый корпорацией Microsoft, называется MIDL), который создает, среди прочего, заголовочный файл, применяемый для компиляции проекта. Компилятор MIDL также создает такие компоненты, как библиотеки типов, которые в UMDF не применяются. Файлы IDL не являются абсолютно необходимыми, т. к. достаточно лишь одного заголовочного файла, содержащего соответствующие объявления. Но они предоставляют структурированный способ для определения интерфейсов и ассоциированных типов данных.

В UMDF файл IDL `Wudfddi.idl` содержит элементы для интерфейсов, которые может предоставлять среда исполнения или которые могут быть реализованы драйверами UMDF. Файл `Wudfddi.idl` содержит определения типов для структур и перечислений, используемых методами UMDF, и объявления для всех интерфейсов UMDF. Для драйверов UMDF обычно не требуются собственные файлы IDL, т. к. для них не нужны библиотеки типов и также потому, что интерфейсы обратного вызова, которые они должны реализовывать, уже объявлены в файлах IDL UMDF.

На вашем компьютере файл Wudfddi.idl должен находиться в папке %wdk%\НомерСборки\inc\wdl\umdf\НомерВерсии.

Довольно часто удобнее получить информацию об объекте или интерфейсе из файла IDL, нежели исследуя соответствующий заголовочный файл. Для примера, рассмотрим исходный код, показанный в листинге 18.1.

**Листинг 18.1. Объявление объекта IWDFObject в файле Wudfddi.idl**

```
[object,
uuid(64275C66-2E71-4060-B5F4-3A76DF96ED3C),
helpstring("IWDFObject Interface"),
local,
restricted,
pointer_default(unique)
]
interface IWDFObject : IUnknown
{
    HRESULT
        DeleteWdfObject(
            void
        );

    HRESULT
        AssignContext(
            [in, unique, annotation("in_opt")]
                IOBJECTCleanup * pCleanupCallback,
            [in, unique, annotation("in_opt")]
                void * pContext
        );

    HRESULT
        RetrieveContext(
            [out, annotation("out")]
                void ** ppvContext
        );

    void
    AcquireLock(
        void
    );

    void ReleaseLock(
        void
    );
};
```

Заголовок вверху примера содержит несколько атрибутов, включающих следующие:

- ◆ атрибут [object], который идентифицирует интерфейс, как интерфейс COM;
- ◆ атрибут [uuid], который назначает интерфейсу идентификатор IID.

Тело интерфейса подобно эквивалентному объявлению в заголовочном файле C++ и содержит следующие элементы:

- ◆ интерфейсы, от которых наследует данный интерфейс;
- ◆ список методов, составляющих интерфейс;

- ◆ типы данных возвращаемых значений и параметров для каждого метода;
- ◆ комментарий для каждого параметра, предоставляющий различного рода информацию.

Например, комментарии `_in`, `_out` и `_inout` указывают направление параметра.

Также в комментариях указывается, является ли параметр обязательным, или же его значение может быть `NULL`. В последнем случае к комментарию направления параметра добавляется `_opt`, например, `_in_opt`.

Дополнительную информацию о том, как интерпретировать содержимое файлов IDL, см. в разделе MSDN **The Interface Definition Language (IDL) File** (Файл языка описания интерфейсов) по адресу <http://go.microsoft.com/fwlink/?LinkId=80074>.

## Использование объектов COM UMDF

Программирование в COM имеет два аспекта: реализация клиентов, использующих существующие объекты COM, и реализация собственных объектов COM. Эти два аспекта не являются взаимоисключающими, т. к. одни объекты COM часто являются клиентами других объектов COM. Как драйверы UMDF, так и среда исполнения UMDF функционируют в качестве клиентов COM.

- ◆ Драйверы UMDF используют предоставляемые UMDF объекты COM для взаимодействия со средой исполнения UMDF.

Например, объект устройства, предоставляемый UMDF, представляет устройство. Драйверы являются клиентами этого инфраструктурного объекта устройства, с помощью которого они выполняют такие задачи, как установка или получение состояния Plug and Play устройства.

- ◆ Среда исполнения UMDF использует предоставляемые драйверами объекты обратного вызова COM для взаимодействия с драйверами.

Например, для обработки запросов ввода/вывода драйвер может создать один или несколько объектов обратного вызова. С помощью этих объектов среда исполнения передает запросы драйверу для обработки.

В этом разделе представлен краткий обзор, каким образом клиенты COM используют существующие объекты COM. Это простейший вид программирования для COM, и ознакомление с этим материалом будет хорошей подготовкой к рассмотрению темы реализации объектов, которая следует дальше в этой главе.

## Использование объекта COM

Чтобы начать использовать объект COM, необходимо получить указатель, по крайней мере, на один из интерфейсов этого объекта. После этого можно вызывать любые методы этого интерфейса, используя такой же синтаксис, как и для указателя на метод в языке C++. В примере, показанном в листинге 18.2, идентификатор `pWdfRequest` является указателем на интерфейс `IWDFIoRequest`.

### Листинг 18.2. Вызов метода интерфейса

```
HRESULT hr;
. . . // Код опущен для краткости.
```

```
hr = pWdfRequest->Send(m_pIUsbTargetDevice,
    WDF_REQUEST_SEND_OPTION_SYNCHRONOUS,
    0);
```

Интерфейс объекта UMDF можно получить одним из следующих способов:

- ◆ среда исполнения UMDF передает указатель интерфейса объекта в качестве входного параметра одному из методов обратного вызова драйвера;
- ◆ драйвер создает объект, вызывая инфраструктурный метод создания объекта;
- ◆ драйвер вызывает метод `IUnknown::QueryInterface`, чтобы запросить новый интерфейс у существующего объекта UMDF.

### **Получение интерфейса через метод обратного вызова**

Это самый простой способ получения интерфейса. Например, когда среда исполнения UMDF вызывает метод `IDriverEntry::OnDeviceAdd` драйвера, она передает ему указатель на интерфейс `IWDFDriver` объекта устройства в качестве входного параметра. Пример этого метода показан в листинге 18.3.

#### **Листинг 18.3. Получение указателя интерфейса через метод обратного вызова**

```
HRESULT CMyDriver::OnDeviceAdd(
    _in IWDFDriver *FxWdfDriver,
    _in IWDFDeviceInitialize *FxDeviceInit)
{
    // Устанавливаем драйвер в стеке устройств.
}
```

После этого реализация метода `OnDeviceAdd` использует указатель `FxWdfDriver` для доступа к методам интерфейса `IWDFDriver` объекта драйвера. Так как указатель интерфейса `IWDFDriver` передается в качестве входного параметра, драйвер не должен освобождать его. Вызывающий клиент обеспечивает, что ассоциированный объект продолжает оставаться действительным на протяжении вызова метода, и освобождает указатель интерфейса, когда он больше не нужен.

### **Получение интерфейса через создание объекта UMDF**

Иногда драйвер должен явно создать объект UMDF, вызывая соответствующий метод создания объекта. Как показано в листинге 18.4, для создания объекта запроса ввода/вывода драйвер вызывает метод `IWDFDevice::CreateRequest` объекта устройства UMDF.

#### **Листинг 18.4. Объявление метода `IWDFDevice::CreateRequest`**

```
HRESULT CreateRequest(
    _in IUnknown* pCallbackInterface,
    _in IWDFObject* pParentObject,
    _out IWDFIoRequest** ppRequest);
```

Идентификатор `ppRequest` является выходным параметром, который предоставляет адрес, по которому метод `CreateRequest` может сохранить указатель на созданный интерфейс

IWDFObject объекта запроса. В следующей процедуре и образце кода показан вызов метода CreateRequest образцом драйвера Fx2\_Driver.

Для обработки параметров при вызове метода CreateRequest:

1. Объявляется переменная pWdfRequest для хранения указателя на интерфейс IWDFIoRequest.
2. Методу CreateRequest передается указатель на переменную pWdfRequest. По возвращению управления методом CreateRequest, переменная pWdfRequest содержит указатель на интерфейс IWDFIoRequest.
3. Переменная pWdfRequest используется для вызова методов интерфейса.

Когда драйверу больше не нужен указатель интерфейса, он должен освободить его, вызвав метод `IUnknown::Release`. Метод получения интерфейса посредством создания объекта (на примере драйвера Fx2\_Driver) показан в листинге 18.5.

#### Листинг 18.5. Получение интерфейса созданием объекта

```
IWDFIoRequest *pWdfRequest = NULL;
. . . // Код опущен для краткости.
hr = m_FxDevice->CreateRequest(NULL,
                                NULL,
                                &pWdfRequest);
. . . // Код опущен для краткости.
hr = pWdfRequest->Send(m_pIUsbTargetDevice,
                        WDF_REQUEST_SEND_OPTION_SYNCHRONOUS,
                        0); // Тайм-аут.
```

### Получение интерфейса через вызов метода `QueryInterface`

Объекты UMDF могут предоставлять несколько интерфейсов. Если имеется указатель на один интерфейс и необходимо получить указатель на другой интерфейс того же самого объекта, требуемый указатель можно получить, вызвав метод `IUnknown::QueryInterface`. В качестве параметров методу передается идентификатор IID требуемого интерфейса и адрес памяти для хранения возвращаемого методом требуемого указателя. Когда интерфейс больше не нужен, его необходимо освободить.

На примере из драйвера Fx2\_Driver, приведенном в листинге 18.6, показано, как драйвер запрашивает указатель на интерфейс `IWDFIoTargetStateManagement` у объекта получателя ввода/вывода UMDF. В примере применяется макрос `IID_PPV_ARGS` (объявленный в файле Objbase.h), который принимает указатель интерфейса и возвращает правильные аргументы для метода `QueryInterface`.

#### Листинг 18.6. Получение нового интерфейса у существующего объекта

```
VOID CMyDevice::StartTarget(IWDFIoTarget * pTarget)
{
    IWDFIoTargetStateManagement * pStateMgmt = NULL;
    HRESULT hrQI = pTarget->QueryInterface(IID_PPV_ARGS(&pStateMgmt));
    . . . // Код опущен для краткости.
}
```

### Полезная информация

Метод `QueryInterface` предоставляемся интерфейсом `IUnknown`. Но, как показано в предыдущем примере, для вызова этого метода не требуется иметь явный указатель на

интерфейс `IUnknown` объекта. Все интерфейсы наследуют от `IUnknown`, поэтому для вызова `QueryInterface` можно использовать любой метод.

## Управление временем жизни объекта COM

Одно из больших отличий между COM и C++ заключается в том, каким образом клиент управляет временем жизни используемого им объекта. В C++ клиенты управляют временем жизни объектов прямым образом, создавая объекты с помощью оператора `new` и уничтожая их оператором `delete`. В отличие от C++, в COM клиенты управляют временем жизни объектов косвенно, посредством подсчета их ссылок.

Работа механизма подсчета ссылок обычно заключается в подсчете числа указателей интерфейса, которые были запрошены и успешно получены, минус число освобожденных указателей. Когда значение счетчика ссылок становится нулевым, что указывает на отсутствие активных интерфейсов, объект самоуничтожается. Клиенты обычно не знают значение счетчика ссылок на объект и не стремятся его узнать. Задача клиента — правильно использовать методы `AddRef` и `Release` и дать объекту обслуживать себя самому. Подробности о том, как правильно использовать методы `AddRef` и `Release`, см. в разд. "Рекомендации по использованию методов `AddRef` и `Release`" ранее в этой главе.

### Полезная информация

Как метод `AddRef`, так и метод `Release` возвращают текущее значение счетчика ссылок объекта. Но на это значение не следует полагаться, оно предоставляется в основном для целей отладки. Настоящее значение счетчика может уже измениться к тому времени, когда вы используете возвращенное значение.

## Реализация инфраструктуры DLL

В дампе именованных экспортруемых компонентов DLL, содержащей внутрипроцессные объекты COM, обычно можно увидеть очень немного экспортруемых функций и ни одного метода, ассоциированного с объектами COM. Причиной этому есть то обстоятельство, что объекты COM и их методы не экспортруются, как в случае с этими элементами в обычной DLL. По существу, одна из точек зрения на COM заключается в том, чтобы рассматривать его как другой, более структурированный способ предоставления доступа к методам DLL.

Для управления процессом DLL должна предоставлять определенную базовую инфраструктуру. Эта инфраструктура состоит из следующих трех основных компонентов:

- ◆ функции `DllMain`, которая предоставляет точку доступа для DLL;
- ◆ одной или нескольких фабрик классов, используемых внешними клиентами для создания экземпляров объектов COM в DLL;
- ◆ экспортруемой по имени функции `DllGetClassObject`, которая предоставляет внешним клиентам экземпляры объектов фабрики класса DLL.

Для стандартных объектов COM также необходимо реализовать и экспортовать по имени функцию `DllCanUnloadNow` и, необязательно, функции `DllRegisterServer` и `DllUnregisterServer`.

Для драйверов UMDF эти функции экспортовать не обязательно.

В этом разделе рассматривается реализация базовой инфраструктуры, необходимой для поддержки драйверов UMDF.

Для реализации большинства драйверов хорошей отправной точкой является использование образцов драйверов Fx2\_Driver или Skeleton, которые затем можно модифицировать под требования определенного драйвера. Самое большое, что может потребоваться, чтобы приспособить данные образцы драйверов к требованиям определенного драйвера, так это внесение в них лишь незначительных модификаций. Код для этих драйверов находится в файлах Corn-sup.cpp и Dllsup.cpp.

## Функция *DllMain*

Библиотека DLL может содержать любое число внутрипроцессных объектов COM, но она должна иметь лишь одну точку входа, которая называется *DllMain*. Windows вызывает функцию *DllMain* после загрузки двоичного файла драйвера в хост-процесс и потом опять перед его выгрузкой. Эта функция также вызывается при создании и уничтожении потоков. Цель вызова функции указывается в параметре *dwReason*.

- ◆ При вызове функции *DllMain* драйвера UMDF для загрузки или выгрузки драйвера функция должна выполнять только простые, глобальные для модуля, задачи инициализации и завершения.

Например, она может инициализировать или освобождать глобальные параметры либо регистрировать и удалять регистрацию трассировки WPP. Реализация функции *DllMain* для драйвера UMDF определенно не должна выполнять такие операции, как вызов функции *LoadLibrary*. Дополнительную информацию о том, какие операции можно выполнять в функции *DllMain*, а какие нельзя, см. в справочном материале для функции в документации к Platform SDK.

- ◆ Когда функция *DllMain* драйвера UMDF вызывается для создания или уничтожения потока, она может игнорировать вызов.

Дополнительную информацию по функции *DllMain* см. в справочном материале для нее в документации Platform SDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80069>.

В листинге 18.7 приводится функция *DllMain* образца драйвера Fx2\_Driver. Она инициализирует и очищает трассировку WPP при загрузке и выгрузке DLL соответственно. Вызовы для создания и уничтожения потоков функция игнорирует.

### Листинг 18.7. Типичная реализация функции *DllMain*

```
BOOL WINAPI DllMain(HINSTANCE ModuleHandle,
                      DWORD Reason,
                      PVOID)
{
    UNREFERENCED_PARAMETER(ModuleHandle);
    if (DLL_PROCESS_ATTACH == Reason) { // Инициализация трассировки.
        WPP_INIT_TRACING(MYDRIVER_TRACING_ID);
    }
    else if (DLL_PROCESS_DETACH == Reason) { // Очистка трассировки.
        WPP_CLEANUP();
    }
    return TRUE;
}
```

## Функция *DllGetClassObject*

Так как фабрики классов не экспортируются по имени, получить прямой доступ к фабрике классов клиент не может. Вместо этого DLL экспортирует по имени функцию *DllGetClassObject*, которую может вызывать любой клиент, загрузивший DLL. Для многих DLL COM, включая образцы драйверов UMDF, функция *DllGetClassObject* является единственной функцией, включенной в DEF-файл проекта для экспортирования по имени из DLL.

Когда клиент хочет создать экземпляр одного из объектов COM в DLL, он передает функции *DllGetClassObject* идентификатор CLSID требуемого объекта фабрики классов вместе с идентификатором IID требуемого интерфейса, обычно *IClassFactory*. Функция *DllGetClassObject* создает новый объект фабрики классов и возвращает указатель на запрошенный интерфейс. Клиент может потом использовать метод *IClassFactory::CreateInstance* для создания экземпляра ассоциированного объекта COM.

Драйверы UMDF получают указатели на объекты UMDF непосредственно из среды исполнения UMDF, поэтому им обычно не требуется вызывать функцию *DllGetClassObject* или применять фабрику классов. Но среда исполнения UMDF должна использовать фабрику классов в начале процесса загрузки, чтобы создать объект обратного вызова драйвера. Это означает, что все драйверы UMDF должны реализовывать функцию *DllGetClassObject*, чтобы предоставить указатель на фабрику классов для объекта обратного вызова драйвера.

В листинге 18.8 приводится реализация функции *DllGetClassObject* образца драйвера *Fx2\_Driver*. Функция получает идентификаторы CLSID и IID и выполняет следующие операции:

1. Определяет, какую фабрику классов необходимо создать.

Для драйверов UMDF требуется только один объект COM с фабрикой классов: объект обратного вызова драйвера. Поэтому реализация просто проверяет на правильность идентификатора CLSID.

2. С помощью оператора *new* языка C++ создает экземпляр фабрика классов для объекта обратного вызова драйвера.

Конструктор объекта устанавливает значение счетчика ссылок равное единице.

3. Вызывает метод *QueryInterface* созданного объекта фабрики классов, чтобы запросить интерфейс, указанный в параметре *InterfaceId*.

Для UMDF всегда запрашивается интерфейс *IClassFactory*. В результате этой операции значение счетчика ссылок увеличивается до двух.

4. Освобождает указатель фабрики классов.

Этот указатель применяется только для временных внутренних нужд и не используется повторно, поэтому он освобождается сразу же, как только отпадает надобность в нем. Теперь значение счетчика ссылок объекта обратного вызова драйвера снова возвратилось к единице, означая один активный интерфейс.

После этого среда исполнения UMDF с помощью интерфейса *IClassFactory* создает экземпляр объекта обратного вызова драйвера. Для большинства драйверов UMDF этот код можно использовать, выполнив незначительные, или вообще никакие, модификации. В листинге 18.8 приведен пример реализации функции *DllGetClassObject*, взятый из образца драйвера *Fx2\_Driver*.

**Листинг 18.8. Типичная реализация функции DllGetClassObject**

```
HRESULT STDMETHODCALLTYPE DllGetClassObject(_in REFCLSID ClassId,
                                         _in REFIID InterfaceId,
                                         _deref_out LPVOID *Interface)

{
    PCClassFactory factory;
    HRESULT hr = S_OK;
    *Interface = NULL;

    if (IsEqualCLSID(ClassId, CLSID_MyDriverCoClass) == false) {
        return CLASS_E_CLASSNOTAVAILABLE;
    }

    factory = new CClassFactory();
    if (NULL == factory) {
        hr = E_OUTOFMEMORY;
    }

    if (SUCCEEDED(hr)) {
        hr = factory->QueryInterface(InterfaceId, Interface);
        factory->Release();
    }

    return hr;
}
```

## Фабрика классов

Некоторые объекты COM должны создаваться непосредственно внешними клиентами и должны иметь фабрику классов. *Фабрика классов* — это небольшой специализированный объект COM, чье единственное назначение состоит в создании нового экземпляра ассоциированного с ним объекта COM и возвращении указателя на указанный интерфейс.

Драйверы UMDF имеют только одного внешнего клиента — среду исполнения, которая использует фабрику классов для создания объекта обратного вызова драйвера. Этот объект функционирует в качестве точки входа для драйвера UMDF по большому счету таким же образом, как и процедура `DriverEntry` для драйверов режима ядра. Для других объектов обратного вызова, таких как, например, объектов обратного вызова очереди или устройства, фабрики классов не требуются, и они рассматриваются далее в этой главе.

Так как среда исполнения UMDF непосредственно создает только один объект обратного вызова, драйверу UMDF требуется лишь одна фабрика классов. Кроме интерфейса `IUnknown`, эта фабрика предоставляет только один интерфейс: `IClassFactory`. Интерфейс `IClassFactory` имеет два члена:

- ◆ метод `CreateInstance` создает экземпляр объекта и возвращает запрошенный интерфейс клиенту;
- ◆ метод `LockServer` удерживает DLL открытой в памяти. В драйверах UMDF этот метод обычно реализован только символически, т. к. в UMDF он не используется.

## Реализация фабрики классов

Фабрика классов обычно реализуется в одном классе C++. В листинге 18.9 приводится пример объявления фабрики классов. Этот код адаптирован из объявления фабрики классов `CClassFactory` объекта обратного вызова драйвера для образца драйвера `Fx2_Driver`. В при-

мере опущены некоторые закрытые члены интерфейса `IUnknown` и показана только часть объявления, имеющая отношение к `IClassFactory`. Полное объявление фабрики можно найти в образце драйвера `Fx2_Driver`.

#### Листинг 18.9. Пример объявления фабрики классов

```
class CClassFactory : public CUnknown, public IClassFactory
{
public:
    . . . // Код опущен для краткости.
    // Методы IClassFactory.
    virtual HRESULT STDMETHODCALLTYPE CreateInstance(
        _in_opt IUnknown *OuterObject,
        _in REFIID InterfaceId,
        _out PVOID *Object);
    virtualHRESULT STDMETHODCALLTYPE LockServer(
        _in BOOL Lock);
};
```

Класс `CClassFactory` наследует от `IClassFactory`. Как и все объекты COM, он также наследует от `IUnknown`, хотя в данном случае это косвенное наследование, через класс `CUnknown`. Эта подробность реализации образца драйвера `Fx2_Driver` рассматривается в разд. "Реализация интерфейса `IUnknown`" далее в этой главе.

В листинге 18.10 приводится реализация метода `IClassFactory::CreateInstance` в образце драйвера `Fx2_Driver`. Это типичная реализация этого метода, и большинство драйверов UMDF могут использовать его без каких-либо дополнительных модификаций. Когда среда исполнения UMDF вызывает этот метод:

- ◆ она игнорирует первый параметр. Этот параметр служит для поддержки агрегирования COM, которое не поддерживается в UMDF;
- ◆ присваивает параметру `InterfaceId` значение идентификатора IID для `IDriverEntry`, т. е. `IID_IDriverEntry`.

#### Листинг 18.10. Реализация метода `IClassFactory::CreateInstance` в образце драйвера `Fx2_Driver`

```
HRESULT STDMETHODCALLTYPE CClassFactory::CreateInstance(
    _in_opt IUnknown * /* Outer-Object */,
    _in REFIID InterfaceId,
    _out PVOID *Object)
{
    HRESULT hr;
    PCMyDriver driver;
    *Object = NULL;
    hr = CMyDriver::CreateInstance(&driver);
    if (SUCCEEDED(hr)) {
        hr = driver->QueryInterface(InterfaceId, Object);
        driver->Release();
    }
    return hr;
}
```

Метод `IClassFactory::CreateInstance` выполняет следующие операции:

1. Создает новый объект обратного вызова драйвера, вызывая для этого метод `CreateInstance` объекта обратного вызова драйвера.

Объекты создаются с помощью оператора языка C++ `new`. В данном случае, объект обратного вызова драйвера реализуется в виде класса с именем `CMyDriver`, который содержит статический метод для создания объектов: `CMyDriver::CreateInstance`. Этот метод создает экземпляр класса с помощью оператора `new` языка C++ и возвращает указатель на этот объект. Конструктор объекта увеличивает значение счетчика ссылок, которое становится равным единице.

2. Запрашивает указатель на интерфейс `IDriverEntry` объекта обратного вызова драйвера, вызывая для этого метод `QueryInterface` объекта.

При успешном исполнении метод `QueryInterface` присваивает параметру `Object` интерфейс `IDriverEntry` объекта обратного вызова драйвера и увеличивает значение счетчика ссылок, вызывая для этого метод `AddRef` объекта. Теперь значения счетчика ссылок объекта равно двум.

3. Вызывает метод `Release` объекта обратного вызова драйвера, чтобы освободить указатель объекта. Это уменьшает значение счетчика ссылок объекта к единице.

4. Возвращает результаты вызова метода `QueryInterface` среде исполнения UMDF.

В случае успеха, переменной `hr` присваивается значение `S_OK`, а среда исполнения UMDF получает указатель на интерфейс `IDriverEntry` объекта обратного вызова драйвера.

В UMDF метод `IClassFactory::LockServer` не применяется, но для удовлетворения требований COM он должен быть реализован хотя бы символически. Реализация метода `LockServer` в образце драйвера (листинг 18.11) просто отслеживает запросы на блокировку и разблокировку с помощью статической переменной.

#### Листинг 18.11. Реализация метода `IClassFactory::LockServer` в образце драйвера Fx2\_Driver

```
HRESULT STDMETHODCALLTYPE CClassFactory::LockServer(_in BOOL Lock)
{
    if (Lock)
    {
        InterlockedIncrement(&s_LockCount);
    }
    else
    {
        InterlockedDecrement(&s_LockCount);
    }
    return S_OK;
}
```

### Объекты, не требующие фабрики классов

Если внешний клиент непосредственно не создает объектов COM, то ему не требуется фабрика классов. Способ создания объектов таким клиентом является деталью реализации клиента. Например, среда исполнения непосредственно не создает объектов обратного вызова устройства. Она вызывает метод `IDriverEntry::OnDeviceAdd` объекта обратного вызова драйвера, который и создает объект обратного вызова устройства. Драйвер потом передает интерфейс

`IUnknown` объекта обратного вызова устройства среди исполнения UMDF, когда он вызывает метод `IWDFDriver::CreateDevice`.

В образце драйвера `Fx2_Driver` метод `OnDeviceAdd` создает объект обратного вызова устройства, вызывая метод `CMyDevice::CreateInstance`. Этот метод создает объект обратного вызова устройства и передает его методу `IWDFDriver::CreateDevice`. В действительности, в образце драйвера эта операция выполняется косвенно в методе `CMyDevice::Initialize`, который вызывается методом `CreateInstance`.

## Реализация объектов обратного вызова UMDF

В основном драйвер UMDF состоит из группы объектов обратного вызова COM, которые реагируют на извещения от среды исполнения UMDF и позволяют драйверу обрабатывать такие события, как запросы на чтение и запись. Все объекты обратного вызова являются внутрипроцессными объектами COM. Основные требования к реализации объектов обратного вызова относительно простые:

- ◆ реализуются методы интерфейса `IUnknown` для обработки подсчета ссылок и предоставляются указатели на интерфейсы объекта;
- ◆ реализуются интерфейсы обратного вызова UMDF, которые объект предоставляет.

В этом разделе рассматриваются основы реализации объектов обратного вызова для драйверов UMDF.

## Реализация класса для объекта COM

Интерфейсы не реализуются сами по себе; они должны предоставляться каким-либо объектом. В типичном подходе к созданию сравнительно простого объекта COM объект реализуется в виде класса на языке C++, который содержит код для поддержки интерфейса `IUnknown` и интерфейсы UMDF, предоставляемые объектом. Таким образом, реализация объекта COM в большой степени сводится к реализации методов, составляющих его интерфейс. При этом необходимо принять во внимание следующие обстоятельства.

- ◆ Класс должен наследовать от всех интерфейсов, которые он предоставляет. Но это требование может выполняться косвенно, наследованием от родительского класса, который в свою очередь наследует от одного или нескольких требуемых интерфейсов. Например, многие из объектов обратного вызова образцов драйверов UMDF не наследуют непосредственно от интерфейса `IUnknown`. Вместо этого они наследуют от класса `CUnknown`, который наследует от интерфейса `IUnknown`.
- ◆ Интерфейсы объявляются как абстрактные базовые классы, поэтому класс должен реализовывать все методы интерфейса.
- ◆ Кроме наследования от интерфейса, класс может наследовать от родительского класса.
- ◆ Класс также может содержать закрытые элементы данных, открытые методы, не являющиеся частью интерфейса, и т. п. Эти компоненты предназначены для внутреннего использования и невидимы для клиентов.
- ◆ Конструкторы нужно применять только для выполнения инициализации членов класса и любую другую инициализацию, которая не может завершиться неудачей.

Любой код, выполнение которого может завершиться неудачей, следует размещать в открытом методе для инициализации, который можно вызвать после создания объекта.

Примером такого метода может служить метод `CMYDevice::Initialize` драйвера `Fx2_Driver`, который находится в файле `Device.cpp`.

Методы интерфейса реализуются как открытые методы класса. В листинге 18.12 приводится схематическое объявление класса `CMYObject`, который реализует интерфейс `IUnknown` и два дополнительных интерфейса: `IWDF1` и `IWDF2`.

#### Листинг 18.12. Типичное объявление класса для простого объекта COM

```
class CMYObject : public IUnknown, public IWDF1, public IWDF2
{
private:
    // Закрытые сервисные методы и данные.

public:
    // Методы интерфейса IUnknown.
    virtual ULONG STDMETHODCALLTYPE AddRef(VOID);
    virtual ULONG STDMETHODCALLTYPE Release(VOID);
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        _in REFIID InterfaceId,
        _out PVOID *Object);
    // Методы интерфейса IWDF1.
    . . . // Код опущен для краткости.
    // Методы интерфейса IWDF2.
    . . . // Код опущен для краткости.
};
```

## Реализация интерфейса `IUnknown`

Интерфейс `IUnknown` является центральным интерфейсом COM, который предоставляется всеми объектами COM и является необходимым для работы объекта. Интерфейс `IUnknown` можно реализовать несколькими способами. В этой главе рассматривается подход, применяемый в образцах драйверов UMDF. А именно:

- ◆ класс `CUnknown` реализует базовую версию интерфейса `IUnknown`;
- ◆ большая часть объекта реализуется отдельным классом, который наследует от `CUnknown` и содержит специфические для интерфейса реализации методов интерфейса `IUnknown`.

Информацию о других подходах к реализации интерфейса `IUnknown` см. в книге "Inside COM и Inside OLE" (издательство Microsoft Press), а также документацию по COM в MSDN.

В листинге 18.13 приведен пример объявления класса `CUnknown`, адаптированный из образца драйвера `Fx2_Driver`. В примере показаны только абсолютно необходимые части класса.

#### Листинг 18.13. Объявление класса для реализации интерфейса `IUnknown`

```
class CUnknown : public IUnknown
{
private:
    LONG m_ReferenceCount; // Подсчет ссылок.
public:
    virtual ULONG STDMETHODCALLTYPE AddRef(VOID);
    virtual ULONG STDMETHODCALLTYPE Release(VOID);
```

```

virtual HRESULT STDMETHODCALLTYPE QueryInterface(
    _in REFIID InterfaceId,
    _out PVOID *Object);
};

}

```

В листинге 18.14 приведен отредактированный код объявления класса для объекта обратного вызова драйвера образца драйвера Fx2\_Driver, в котором показываются методы интерфейса `IUnknown`.

#### Листинг 18.14. Объявление класса для объекта обратного вызова драйвера Fx2\_Driver

```

class CMyDriver : public CUnknown, public IDriverEntry
{
private:
    . . . // Код опущен для краткости.
public:
    // Реализация интерфейса IUnknown, специфическая для IDriverEntry.
    virtual ULONG STDMETHODCALLTYPE AddRef(VOID);
    virtual ULONG STDMETHODCALLTYPE Release(VOID);
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        _in REFIID InterfaceId,
        _out PVOID *Object);
    // Реализация IDriverEntry.
    . . . // Код опущен для краткости.
};

```

### Методы `AddRef` и `Release`

Подсчет ссылок, возможно, является ключевой задачей интерфейса `IUnknown`. Обычно подсчет ссылок ведется для объекта в целом, хотя методы `AddRef` и `Release` могут вызываться с любого интерфейса. В образцах драйверов это требование выполняется посредством передачи вызовов, выдаваемых специфическими для интерфейса реализациями, базовой реализации, и через предоставление этим методам заниматься увеличением и уменьшением элемента данных, содержащего счетчик ссылок. Таким образом, все запросы `AddRef` и `Release` обрабатываются в одном месте, уменьшая возможность ошибок.

В листинге 18.15 приведен пример кода типичной реализации метода `AddRef`, взятый из базовой реализации интерфейса `IUnknown` в образце драйвера Fx2\_Driver. Переменная `m_ReferenceCount` — это закрытый элемент данных, в котором хранится значение счетчика ссылок. Обратите внимание на применение процедуры `InterlockedIncrement` вместо оператора `++`. Процедура `InterlockedIncrement` блокирует счетчик ссылок при его увеличении, таким образом исключая возможность возникновения проблем, порождаемых состоянием гонок.

#### Листинг 18.15. Базовая реализация метода `AddRef` в образце драйвера Fx2\_Driver

```

ULONG STDMETHODCALLTYPE CUnknown::AddRef(VOID)
{
    return InterlockedIncrement(&m_ReferenceCount);
}

```

Метод `Release` похож на метод `AddRef`, но чуть посложнее. Этот метод уменьшает значение счетчика ссылок, после чего проверяет, не равно ли оно нулю. Нуевое значение счетчика

ссылок означает, что больше нет активных интерфейсов, и метод `Release` уничтожает объект с помощью оператора `delete` языка C++. В листинге 18.16 приведен пример реализации метода `Release`, взятый из образца драйвера `Fx2_Driver`.

#### Листинг 18.16. Базовая реализация метода `Release` в образце драйвера `Fx2_Driver`

```
ULONG STDMETHODCALLTYPE CUnknown::Release(VOID)
{
    ULONG count = InterlockedDecrement(&m_ReferenceCount);
    if (count == 0) {
        delete this;
    }
    return count;
}
```

Как метод `AddRef`, так и метод `Release` возвращают текущее значение счетчика ссылок, что иногда полезно при отладке.

Реализации методов `AddRef` и `Release`, специфические для интерфейса, просто используют ключевое слово `_super`, чтобы вызвать базовую реализацию метода и возвратить значение метода. В листинге 18.17 приведена специфическая для конкретного интерфейса реализация метода `AddRef`. Реализация метода `Release` подобна реализации метода `AddRef`.

#### Листинг 18.17. Реализация метода `AddRef`, специфическая для интерфейса

```
ULONG STDMETHODCALLTYPE CMyDriver::AddRef(VOID)
{
    return _super::AddRef();
}
```

### Функция `QueryInterface`

Функция `QueryInterface` представляет собой фундаментальный механизм, с помощью которого объект COM предоставляет указатели на свои интерфейсы. Она отвечает на запросы клиентов, возвращая указатель на требуемый интерфейс. В листинге 18.18 приведен пример типичной реализации функции `QueryInterface`, взятый из базовой реализации интерфейса `IUnknown` в образце драйвера `Fx2_Driver` и слегка модифицированный.

#### Листинг 18.18. Базовая реализация функции `QueryInterface` в образце драйвера `Fx2_Driver`

```
HRESULT STDMETHODCALLTYPE CUnknown::QueryInterface(
    _in REFIID InterfaceId,
    _out PVOID *Interface)
{
    if (IsEqualIID(InterfaceId, _uuidof(IUnknown))) {
        AddRef();
        *Interface = static_cast<IUnknown*>(this);
        return S_OK;
    }
    else {
        *Interface = NULL;
    }
}
```

```

        return E_NOINTERFACE;
    }
}
}

```

Функция `QueryInterface` сначала проверяет, указывает ли запрашиваемый идентификатор IID поддерживаемый интерфейс. В данном примере поддерживается только один интерфейс — `IUnknown`, поэтому имеется только один действительный идентификатор IID. Для выполнения этой проверки в методе применяются две полезные утилиты:

- ◆ сервисная функция `IsEqualIID`, объявленная в файле `Guiddef.h`. Возвращает `TRUE`, если идентификаторы IID совпадают, и `FALSE` — в противном случае;
- ◆ оператор `_uuidof` для версии языка C++ от корпорации Microsoft. Возвращает идентификатор IID указанного типа интерфейса.

Если запрашиваемый интерфейс поддерживается, функция `QueryInterface` вызывает метод `AddRef`, чтобы увеличить значение счетчика ссылок на объект, и возвращает указатель интерфейса. Чтобы возвратить указатель, функция `QueryInterface` приводит указатель `this` к типу требуемого интерфейса. Данное приведение типов обусловлено способом, в который C++ обрабатывает множественное наследование. Приведение указателя `this` к соответствующему типу интерфейса обеспечивает, что указатель будет находиться в правильном месте в таблице VTable.

Если запрашиваемый идентификатор IID не поддерживается, функция `QueryInterface` присваивает `Interface` значение `NULL` и возвращает стандартное значение ошибки `HRESULT` — `E_NOINTERFACE`.

### Внимание!

Все реализации функции `QueryInterface` объекта должны возвращать одинаковый указатель интерфейса `IUnknown`. Например, если объект предоставляет два интерфейса — `InterfaceA` и `Interfaces` — клиент, вызывающий функцию `QueryInterface` на любом из этих интерфейсов, должен получить тот же самый указатель интерфейса `IUnknown`.

Специфические реализации интерфейса `IUnknown` образцов драйверов UMDF возвращают указатель интерфейса только в том случае, если был запрошен данный конкретный интерфейс. Все другие запросы пересыпаются базовой реализации интерфейса. В листинге 18.19 приведен пример реализации функции `QueryInterface` для интерфейса `IDriverEntry`, взятый из образца драйвера `Fx2_Driver`. Функция возвращает указатель на интерфейс `IDriverEntry`, когда запрашивается данный интерфейс, и пересыпает все другие запросы базовой реализации интерфейса.

#### Листинг 18.19. Базовая реализация функции `QueryInterface` в образце драйвера `Fx2_Driver`

```

HRESULT CMYDriver::QueryInterface(
    _in REFIID InterfaceId,
    _out PVOID *Interface)
{
    if (IsEqualIID(InterfaceId, _uuidof(IDriverEntry))) {
        AddRef();
        *Interface = static_cast<IDriverEntry*>(this);
        return S_OK;
    }
}

```

```

    else {
        return CUnknown::QueryInterface(InterfaceId, Interface);
    }
}
}

```

## Реализация объектов обратного вызова UMDF

Все объекты обратного вызова обычно имеют взаимно-однозначное соотношение с соответствующим объектом UMDF. Например, объект обратного вызова драйвера ассоциирован с объектом драйвера UMDF. Объект обратного вызова предназначен для выполнения двух основных задач:

- ◆ получать извещения от среды исполнения UMDF, связанные с ассоциированным объектом UMDF;
- ◆ предоставлять область контекста для хранения данных контекста для ассоциированного объекта UMDF.

Объекты обратного вызова UMDF предоставляют интерфейс `IUnknown` и, по крайней мере, один добавочный интерфейс UMDF, такой как, например, интерфейс `IDriverEntry` или `IPnPCallback`. Число и тип интерфейсов UMDF, предоставляемых объектом обратного вызова, зависят от конкретного объекта и требований драйвера. Объекты обычно имеют, по крайней мере, один требуемый интерфейс. Например, объект обратного вызова драйвера обязан предоставлять один интерфейс UMDF: `IDriverEntry`.

Многие объекты обратного вызова также имеют один или несколько необязательных интерфейсов, которые реализуются, только если они необходимы для драйвера. Например, в зависимости от требований драйвера, объект обратного вызова устройства может предоставлять несколько необязательных интерфейсов, в том числе интерфейсы `IPnPCallback`, `IPnPCallbackHardware` и `IPnPCallbackSelfManagedIo`.

Большинство деталей реализации интерфейсов UMDF связаны с индивидуальными методами и не обсуждаются в этой книге.

Объект обратного вызова UMDF обычно реализуется в виде класса, который наследует от интерфейса `IUnknown` и от одного или нескольких специфических для объекта интерфейсов. В листинге 18.20 приведено полное объявление класса `CMyDriver`, взятое из объекта обратного вызова драйвера образца драйвера `Fx2_Driver`. Данный класс наследует от одного интерфейса UMDF — `Fx2_Driver`, а также наследует от интерфейса `IUnknown` через родительский класс `CUnknown`. Для удобства нескольких простых методов реализованы в этом классе, а не в файле `associated.cpp`.

### Листинг 18.20. Объявление класса объекта обратного вызова драйвера образца `Fx2_Driver`

```

class CMyDriver : public CUnknown, public IDriverEntry
{
private:
    IDriverEntry * QueryIDriverEntry(VOID)
    {
        AddRef();
        return static_cast<IDriverEntry*>(this);
    }
    HRESULT Initialize(VOID);
public:
    static HRESULT CreateInstance(_out PCMyDriver *Driver);
}

```

```

public:
    virtual HRESULT STDMETHODCALLTYPE OnInitialize(
        _in IWDFDriver *FxWdfDriver)
    {
        UNREFERENCED_PARAMETER(FxWdfDriver);
        return S_OK;
    }
    virtual HRESULT STDMETHODCALLTYPE OnDeviceAdd(
        _in IWDFDriver *FxWdfDriver,
        _in IWDFDeviceInitialize *FxDeviceInit);
    virtual VOID STDMETHODCALLTYPE OnDeinitialize(
        _in IWDFDriver *FxWdfDriver)
    {
        UNREFERENCED_PARAMETER(FxWdfDriver);
        return;
    }
    virtual ULONG STDMETHODCALLTYPE AddRef(VOID)
    {
        return _super::AddRef();
    }
    virtual ULONG STDMETHODCALLTYPE Release(VOID)
    {
        return _super::Release();
    }
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        _in REFIID InterfaceId,
        _deref_out PVOID *Object);
};

};

```

При реализации объектов обратного вызова UMDF, среди прочих, следует принять во внимание следующие аспекты.

- ◆ Классы должны наследовать от всех интерфейсов, которые они предоставляют. Но наследование может быть косвенным, например, наследование от класса, который, в свою очередь, наследует от одного или нескольких интерфейсов.
- ◆ Классы должны реализовывать все методы интерфейсов, которые они предоставляют. Это требование обусловлено тем, что интерфейсы объявляются, как абстрактные базовые классы.
- ◆ Классы также могут содержать закрытые элементы данных, открытые методы, не являющиеся частью интерфейса, и т. п. Эти компоненты предназначены для внутреннего использования объектом и другими объектами в DLL, которые невидимы для клиентов.
- ◆ Классы могут иметь методы доступа для открытого предоставления элементов данных. Хорошим правилом будет применение конструкторов для инициализации членов класса и выполнения любой другой инициализации, которая гарантированно не может завершиться неудачей. Конструкторы не должны содержать код, исполнение которого может завершиться неудачей. Любой код, выполнение которого может завершиться неудачей, следует размещать в открытом методе для инициализации, который можно вызвать после создания объекта. Примером такой функции может служить метод `CMyDevice::Initialize` драйвера `Fx2_Driver`, который находится в файле `Device.cpp`.

# **ЧАСТЬ V**

## **Создание, установка и тестирование драйверов WDF**

**Глава 19.** Сборка драйверов WDF

**Глава 20.** Установка драйверов WDF

**Глава 21.** Инструменты для тестирования драйверов WDF

**Глава 22.** Отладка драйверов WDF

**Глава 23.** Инструмент PREfast for Drivers

**Глава 24.** Инструмент Static Driver Verifier

# ГЛАВА 19

## Сборка драйверов WDF

Для сборки как драйверов UMDF, так и драйверов KMDF используются одинаковые инструменты и процедуры. Хотя WDF поддерживает другой интерфейс DDI и другую модель программирования, основной процесс сборки драйверов WDF все равно по большому счету такой же, как и для других драйверных моделей Windows.

В этой главе дается введение в процесс сборки драйверов для Windows и употребляющихся для этого инструментов для разработчиков, еще не знакомых с этим предметом. Также в ней рассматривается несколько связанных со сборкой вопросов, специфичных для драйверов WDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b>	
Build.exe; Ntddk.h	%wdk%\bin\<amd64   ia64   x86>
Wdf.h	%wdk%\inc\wdf\kmdf
Wudfddi.h	%wdk%\inc\wdf\umdf
<b>Образцы драйверов</b>	
Fx2_Driver's Makefile.inc and Sources	%wdk%\src\umdf\usb\fx_2driver
KMDF Featured Toaster Makefile.inc	%wdk%\src\kmdf\toaster
<b>Документация WDK</b>	
Документация утилиты Build	<a href="http://go.microsoft.com/fwlink/?LinkId=80609">http://go.microsoft.com/fwlink/?LinkId=80609</a>
Раздел "WDK Building and Loading a Framework-based Driver" <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79347">http://go.microsoft.com/fwlink/?LinkId=79347</a>

## Общие положения по сборке драйверов

Сборка драйверов существенным образом отличается от сборки приложений. Для драйверов не существует комплексных средств разработки для автоматизации большинства сложных задач, связанных с конфигурированием и исполнением процесса сборки, подобных инстру-

<sup>1</sup> Сборка и загрузка драйвера на основе инфраструктуры (WDF). — Пер.

ментам для разработки приложений, таких как, например Microsoft Visual Studio. Вместо этого разработчики драйверов действуют следующим образом.

- ◆ Создают и редактируют файлы проекта в текстовом редакторе.  
Обычно применяется специальный редактор для работы с кодом, но можно использовать любой текстовый редактор.
- ◆ Выполняют сборку драйвера с помощью утилиты командной строки Build, которая поставляется вместе с набором разработчика WDK.  
Утилита Build выполняет сборку двоичного файла драйвера и родственных файлов, применяя для этого компиляторы, компоновщики и препроцессоры, также поставляемые с набором разработчика WDK.
- ◆ Кроме исходных файлов, создают файл make и несколько других вспомогательных файлов, содержащие инструкции и данные, которые утилиты Build использует для выполнения сборки.

Все эти файлы нужно создавать вручную в текстовом редакторе. Разработчики обычно создают вспомогательные файлы для проекта, копируя файлы, используемые в подобном образце драйвера и потом модифицируя их под свои требования.

Процесс создания вспомогательных файлов и сборки драйверов подобен для обоих типов драйверов. Тем не менее между процессами для UMDF и KMDF имеются некоторые основные различия, истекающие из выбора языка программирования и того обстоятельства, что в них применяются разные интерфейсы DDI.

## **Драйверы UMDF: обстоятельства, учитывающиеся при сборке**

При подготовке к выполнению сборки драйверов UMDF следует принять во внимание следующие обстоятельства.

- ◆ Драйверы UMDF почти всегда пишутся на языке C++.  
По существу, драйверы UMDF представляют собой коллекцию объектов COM, а язык C++ является предпочтительным языком для разработки программ COM. Хотя разработка драйверов UMDF с помощью языка С является возможной, такой подход очень неудобен и применяется только в редких случаях.
- ◆ UMDF не поддерживает реализацию драйверов с применением языков для управляемого кода, таких как, например, C#.
- ◆ Файлы исходного кода UMDF имеют расширение CPP и обрабатываются с помощью компилятора для языка C++.
- ◆ Среди исходных файлов обязательно должен быть файл Wudfddi.h.  
В этом стандартном заголовочном файле содержатся объявления для интерфейса DDI, используемого драйверами UMDF.
- ◆ Для драйверов должна выполняться привязка к библиотекам UMDF.

## **Драйверы KMDF: обстоятельства, учитывающиеся при сборке**

При подготовке к выполнению сборки драйверов KMDF следует принять во внимание следующие обстоятельства.

- ◆ Для написания драйверов KMDF применяется язык С.

Для разработки приложений режима ядра язык С++ имеет только ограниченное применение. В то время как возможно применение таких базовых возможностей языка С++, как объявление переменных в теле программы, а не в разделе объявлений (inline variable declaration), код, созданный объектно-ориентированными возможностями языка С++, не гарантирует правильную работу в режиме ядра. Например, компилятор может выделить память из неправильного пула.

Дополнительную информацию о применении языка С++ для создания драйверов режима ядра см. в разделе **C++ for Kernel Mode Drivers: Pros and Cons** (Применение языка С++ для разработки драйверов режима ядра: аргументы за и против) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=80060>.

- ◆ Для исходных файлов KMDF можно применять расширение .c, чтобы использовать компилятор для языка С. Но многие проекты компилируются с помощью компилятора языка С++, и поэтому в них применяется расширение .cpp, что является рекомендуемой практикой. Компилятор языка С++ хорошо работает с кодом языка С и предоставляет лучшие возможности обнаружения ошибок и обеспечения контроля типов, чем компилятор языка С. Но чтобы подавить декорирование имен в С++, к объявлениям функций необходимо добавить префикс `extern C`.

- ◆ Среди исходных файлов должны быть файлы Ntddk.h и Wdf.h.

В этих стандартных заголовочных файлах содержатся объявления для интерфейса DDI, используемого драйверами KMDF.

- ◆ Для драйверов должна выполняться привязка к библиотекам KMDF.

Обзор инструментов и процедур для сборки драйверов Windows см. в разделе **Building Drivers** в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79348>.

## Введение в сборку драйверов

Для сборки драйверов WDF применяется утилита Build (Build.exe). Эта утилита командной строки применяется в Microsoft для сборки самой операционной системы Windows. Утилита Build — единственная утилита для сборки драйверов, поддерживаемая корпорацией Microsoft. Но ее можно применять и для сборки других проектов, включая приложения пользовательского режима. В этом разделе дается введение в использование утилиты Build для сборки двоичных файлов драйверов и сопутствующих файлов, таких как INF-файлы.

### Среда сборки

Утилита Build является консольным приложением. Установкой переменных среды для окна консоли задаются следующие три параметра сборки.

- ◆ Версия Windows, под которую разрабатывается драйвер.

Драйверы UMDF можно устанавливать для операционных систем Windows XP, Windows Vista и более поздних версий Windows.

Драйверы KMDF можно устанавливать для операционных систем Windows 2000 и более поздних версий Windows.

- ◆ Процессорная архитектура, под которую разрабатывается драйвер.

Драйверы UMDF можно устанавливать на системах с процессорами архитектуры x86 и x64.

Драйверы KMDF можно устанавливать на все три поддерживающие архитектуры: x86, x64 и Intel Itanium.

- ◆ Проверочная или свободная сборка драйвера.

Проверочные сборки предоставляют обширную поддержку для отладки, но имеют склонность к сравнительно медленной работе. В свободных сборках отсутствует отладочная информация, но они полностью оптимизированы.

Среду сборки можно сконфигурировать вручную в обычной консоли, но при этом можно столкнуться с определенными сложностями. Чтобы упростить процесс установки среды сборки, WDK содержит набор окон консолей среды исполнения с соответствующими настройками среды, по одному окну для каждой комбинации версии Windows, процессорной архитектуры и типа сборки.

### **Полезная информация**

Подробную информацию о настройках среды сборки см. в файле Setenv.bat в каталоге %wdk%\bin. Этот командный файл применяется для конфигурирования определенной среды для окна среды сборки.

#### **Чтобы открыть окно среды сборки:**

1. Нажмите кнопку **Start** (Пуск) на панели задач и выберите команду **All Programs** (Все программы).
2. Выберите пункт меню **Windows Driver Kits**, далее — самую последнюю версию WDK, после чего нажмите пункт меню **Build Environments**.
3. Выберите требуемую процессорную архитектуру, а затем — окно проверочной или свободной среды для требуемой версии Windows.

Кроме указанной версии Windows, окно среды сборки для указанной версии сборки можно применять и для более поздних версий Windows.

#### **Примечание**

Хотя драйверы UMDF можно устанавливать как под Windows Vista, так и под Windows XP, их сборку необходимо выполнять в среде сборки для Windows Vista.

### **Вспомогательные файлы утилиты Build**

Утилита Build зависит от нескольких вспомогательных файлов, в которых указываются подробности выполнения сборки, а также исходные и заголовочные файлы проекта. Все эти вспомогательные файлы представляют собой обычные текстовые файлы, создаваемые вручную в текстовом редакторе. Инструментов для автоматического создания таких файлов не существует. Но для большинства проектов можно скопировать вспомогательные файлы для соответствующего образца драйвера и потом модифицировать их в соответствии с требованиями разрабатываемого драйвера.

## Обязательные файлы

Для всех проектов в обязательном порядке требуются, по крайней мере, следующие два вспомогательных файла.

### ◆ Файл Make.

Этот файл содержит инструкции по выполнению сборки. Он должен называться `Makefile` и содержать лишь следующую директиву, которая подключает стандартный файл `make` WDK — `Makefile.def`:

```
!INCLUDE $(NTMAKEENV) \Makefile.def
```

### Полезная информация

Вы можете сэкономить немного времени на создании файла `make`, просто скопировав его из одного из образцов драйверов WDF.

### ◆ Файл Sources.

Этот файл содержит информацию для сборки драйвера, специфическую для проекта, например, список исходных файлов. В примере из листинга 19.1 показано содержимое базового файла `Sources` для драйвера KMDF под названием `WdfSimple`, указывающего единственный исходный файл — `WdfSimple.c`.

#### Листинг 19.1. Минимальный файл Sources для KMDF

```
TARGETNAME=WdfSimple  
TARGETTYPE=DRIVER  
KMDF_VERSION=1  
SOURCES=WdfSimple.c
```

Более типичный файл `Sources` содержит несколько дополнительных директив. Например, во многих проектах применяются специальные конечные файлы, для которых файл `Sources` должен содержать соответствующие директивы. Далее в этой главе приводятся примеры более сложных файлов `Sources` вместе с объяснениями содержащихся в них директив.

## Необязательные файлы

Проекты драйверов обычно содержат один или несколько следующих факультативных файлов.

### ◆ Файл Dirs.

Этот файл содержит список подпапок для сборки. Он применяется в проектах, в которых исходные файлы размещены в нескольких подпапках, или для сборки нескольких проектов одной командой `build`.

### ◆ Файл INXfile (с расширением `inx`).

Файл INX представляет собой архитектурно-независимый INF-файл. Утилита `Build`, которой задаются соответствующие инструкции, использует данные в файле INX для создания соответствующего файла INF для проекта. Дополнительная информация о файлах INX приводится в главе 20.

### ◆ Файл `Makefile.inc`.

Файл `make` предоставляет поддержку для сборки стандартного конечного файла — двоичного файла драйвера. Но во многих проектах создаются также один или несколько

специальных конечных файлов. Типичным примером таких проектов являются проекты, использующие INX-файлы для автоматического создания INF-файлов, специфических для определенной архитектуры, или драйверов, поддерживающих интерфейс WMI. В таком случае, директивы для сборки специальных конечных файлов помещаются в файл Makefile.inc. Сам файл Makefile никогда нельзя модифицировать.

◆ **Файлы ресурсов (с расширением RE).**

Эти файлы содержат разнообразные ресурсы. Например, ресурс VERSION содержит информацию о номере версии и производителе.

◆ **Файл ресурса MOF (с расширением MOF).**

Драйверы, поддерживающие интерфейс WMI, должны иметь файл ресурса MOF.

### Примечание

Файл Makefile.inc для проектов, содержащих файлы MOF или INX, должен содержать директивы и данные для создания утилитой Build соответствующих выходных файлов. Несколько примеров таких файлов Makefile.inc приводится далее в этой главе.

Информацию о файлах INX см. в справке для консольного инструмента Stampinf.exe, который находится в папке %wdk%\bin. Для вывода справки выполните команду stampinf /? в окне консоли.

По умолчанию утилита Build предполагает, что файлы make, Makefile.inc, Sources и Dirs называются Makefile, Makefile.inc, Sources и Dirs соответственно, поэтому эти имена и используются в большинстве проектов. Но с одним исключением: большинству файлов и папок проекта можно давать любые удобные имена. Исключение — имена файлов не должны содержать пробелов или символов, не входящих в набор ANSI.

Подробности о содержимом и формате этих файлов см. в разделе **Build Utility Reference** (Справка по утилите Build) в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79349>.

### Ограничения на файлы проекта

Утилита Build накладывает некоторые ограничения на организацию проектов. Утилита может работать только с файлами, находящимися в одной папке с файлом Sources, в родительской папке подпапки с файлом Sources или зависящей от платформы подпапке папки с файлом Sources. Если исходные файлы находятся в других, чем эти папках, их необходимо скомпилировать в LIB-файлы с помощью отдельной процедуры сборки, после чего скомпоновать их с LIB-файлами.

Подробную информацию об утилите Build см. в разделе **Build Utility Limitations and Rules** (Правила и ограничения использования утилиты Build) документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80059>.

## Сборка проекта

Процедура сборки проекта не представляет собой ничего сложного.

### Чтобы выполнить сборку проекта:

1. Откройте необходимое консольное окно среды сборки.
2. С помощью команды cd перейдите в папку проекта.

3. Чтобы скомпилировать и скомпоновать драйвер, запустите утилиту Build командой следующего формата:

```
build [-a[b[c]...]]
```

В этой команде набор знаков `[-a[b[c]...]]` представляет аргументы сборки, большинство из которых состоят из одного чувствительного к регистру символа, как описывается в следующем разделе.

## Широко употребляемые флаги утилиты Build

Полный список флагов для утилиты Build можно посмотреть в документации WDK. Некоторые из наиболее употребляемых флагов таковы:

- ◆ ? — выводит список всех флагов;
- ◆ c — удаляет все объектные файлы;
- ◆ e — генерирует файлы журнала, ошибок и предупреждений;
- ◆ g — применяется цветной текст для вывода предупреждений, ошибок и сводок;
- ◆ z — предотвращает проверку или сканирование зависимостей исходных файлов.

Обычно при запуске утилиты сборки задается относительно небольшое число аргументов. Кроме этого, утилита Build предполагает, что проект имеет make-файл с названием Makefile, файл Sources, содержащий список исходных файлов, и т. д. Поэтому при использовании стандартных имен эти файлы не нужно указывать явно. Соответственно, команды для запуска утилиты Build обычно очень простые, как показано в следующем примере:

```
build -ceZ
```

Таким образом, сама команда build относительно проста, т. к. все подробности находятся во вспомогательных файлах, которые рассматриваются в следующих двух разделах.

## Распространенные выходные файлы, создаваемые утилитой Build

Утилита Build может создавать разные типы выходных файлов, которые она обычно помещает в подпапку в папке проекта. Имя папки по умолчанию для выходных файлов зависит от типа среды сборки. Например, папка по умолчанию для выходных файлов для среды Windows XP x86 свободной сборки называется ProjectFolder\objfre\_wxp\_x86\i386.

При любой сборке создаются следующие три типа выходных файлов.

### ◆ Двоичный файл драйвера.

Драйверы KMDF упаковываются как SYS-файлы, а драйверы UMDF — как DLL-файлы. Эти файлы имеют стандартные имена *TargetName.sys* и *TargetName.dll* соответственно.

### ◆ Объектный файл.

Для каждого исходного файла утилиты Build создает объектный файл. Этот файл называется *SourceFileName.obj*.

### ◆ Файл идентификаторов.

Файл идентификаторов драйвера называется *TargetName.pdb*. Этот файл нужен для отладки или генерирования сообщений трассировки.

С помощью утилиты Build при необходимости можно также создавать другие выходные файлы, такие как дополнительные библиотеки DLL или статические библиотеки. Также, если имеются соответствующие файлы Makefile.inc и INX, утилита Build создает в папке для выходных файлов INF-файл, называющийся *TargetName.inf*,

### **Полезная информация**

Для некоторых проектов, например, в которых используются файлы IDL, утилита Build динамически создает заголовочные файлы, помещаемые в папку для выходных файлов. Чтобы разрешить включение заголовка в C- и CPP-файлы, необходимо добавить папку `$ (OBJ_PATH)` в список папок в макросе INCLUDES в файле Sources проекта.

### **Полезные советы для UMDF**

Далее приводится пара рекомендаций по сборке драйверов UMDF.

- ◆ Выдаваемое утилитой Build сообщение об ошибке "Cannot instantiate abstract class" ("Невозможно создать экземпляр абстрактного класса") обычно означает, что один или несколько методов интерфейса не были переопределены. Интерфейсы объявляются как абстрактные базовые классы, поэтому необходимо переопределить все методы интерфейса. Обычно недостающий метод можно определить, явно исполняя утилиту Nmake. Исполнение команды `Nmake all` выведет дополнительную информацию о сборке, включая полные сообщения об ошибках, выданных компилятором, которые позволят определить недостающие методы. Если в проекте используется файл Dirs для выполнения сборки из исходных файлов, расположенных в нескольких подпапках, команду `Nmake all` необходимо исполнить в "листовой" папке (Leaf folder), т. е. папке, содержащей только файл Sources.
- ◆ При объявлении, но не реализованном методе компоновщик выдает сообщение об ошибке, что метод не найден.

## **Пример UMDF: сборка образца драйвера Fx2\_Driver**

В этом разделе приводится пошаговый разбор процесса создания проверочной версии драйвера Fx2\_Driver для Windows Vista и рассматриваются подробности некоторых вспомогательных файлов, упомянутых в предыдущем разделе.

### **Файл Sources для драйвера Fx2\_Driver**

Файл Sources содержит большую часть информации, специфической для проекта, которая используется утилитой Build для сборки проекта. Он состоит из последовательности директив, которые назначают специфические для проекта значения набору макросов и переменных среды. Содержимое файла Sources для драйвера Fx2\_Driver (с удаленными комментариями) показано в листинге 19.2. Это типичный файл Sources для простого драйвера UMDF.

#### **Листинг 19.2. Файл Sources для образца драйвера Fx2\_Driver**

```
UMDF_VERSION=1
UMDF_MINOR_VERSION=5
TARGETNAME=WUDFOsrUsbFx2
```

```
TARGETTYPE=DYNLINK
USE_MSVCRT=1
WIN32_WINNT_VERSION=$(LATEST_WIN32_WINNT_VERSION)
_NT_TARGET_VERSION=$( _NT_TARGET_VERSION_WINXP)
NTDDI_VERSION=$(LATEST_NTDDI_VERSION)
MSC_WARNING_LEVEL=/W4 /WX
MSC_WARNING_LEVEL=$(MSC_WARNING_LEVEL) /wd4201
C_DEFINES = $(C_DEFINES) /D_UNICODE /DUNICODE
DLLENTRY=_DllMainCRTStartup
DLLDEF=exports.def
INCLUDES=$(INCLUDES); .\inc; ..\..\inc
SOURCES=\
    OsrUsbFx2.rc \
    dllsup.cpp \
    comsup.cpp \
    driver.cpp \
    device.cpp \
    queue.cpp \
    ControlQueue.cpp \
    ReadWriteQueue.cpp
TARGETLIBS=\
    $(SDK_LIB_PATH)\strsafe.lib \
    $(SDK_LIB_PATH)\kernel32.lib \
    $(SDK_LIB_PATH)\advapi_32.lib
NTTARGETFILES=$(OBJ_PATH)\$(0)\WUDFOsrUsbFx2.inf
MISCFILES=$(NTTARGETFILES)
RUN_WPP= $(SOURCES) -dll -scan:internal.h
```

В следующем разделе приводится краткое описание макросов и переменных среды, используемых в файле Sources для образца драйвера Fx2\_Driver.

Полный список возможного содержимого файла Sources можно просмотреть в разделе **Utilizing a Sources File Template** (Использование шаблона файла Sources) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79350>. Там же находятся ссылки на документацию о переменных среды и макросах.

## Макросы файла Sources для драйвера Fx2\_Driver

В этом разделе рассматриваются макросы и переменные среды для образца драйвера Fx2\_Driver.

### Номера версии UMDF

Номер версии UMDF определяется следующими макросами.

- ◆ **Макрос UMDF\_VERSION.**  
Обязательный макрос. Указывает номер основной версии UMDF. В примере в листинге 19.2 это версия 1.
- ◆ **Макрос UMDF\_MINOR\_VERSION.**  
Обязательный макрос. Указывает номер дополнительной версии UMDF. В примере в листинге 19.2 это версия 5.

Версии UMDF, определяемые этими макросами, задают следующие аспекты сборки.

- ◆ Заголовки, используемые в драйверах.
- ◆ Основную версию UMDF, к которой выполняется привязка драйвера.

Если в системе пользователя устанавливается новая основная версия инфраструктуры, то старая версия не удаляется, а также остается в системе. Все драйверы, привязанные к старой версии, продолжают использовать ее.

- ◆ Минимальная дополнительная версия UMDF, к которой может быть привязан драйвер.

Можно указать только минимальную дополнительную версию. Если в систему пользователя устанавливается более новая дополнительная версия инфраструктуры, старая версия удаляется, и драйвер автоматически привязывается к новой версии.

- ◆ Требуемый соинсталлятор UMDF.

Версия соинсталлятора должна соответствовать номерам основной и дополнительной версий инфраструктуры.

Дополнительная информация о соинсталляторах WDF и управлении версиями приводится в *главе 20*.

## Стандартные выходные файлы

Следующие макросы описывают стандартные выходные файлы, такие как, например, двоичный файл драйвера.

- ◆ Макрос TARGETNAME.

Обязательный макрос. Указывает имена для таких выходных файлов проекта, как файлы DLL и PDB. В листинге 19.2 применяется значение `WUDFOsrusbfx2`.

- ◆ Макрос TARGETTYPE.

Обязательный макрос. Указывает тип выходного двоичного файла. Драйверы UMDF собираются в виде библиотек DLL, поэтому `TARGETTYPE` устанавливается в `DYNLINK`.

## Специальные выходные файлы

Следующие макросы описывают специальные выходные файлы, такие как INF-файл, создаваемый утилитой Build при обработке INX-файла.

- ◆ Макрос MISCFILES.

Необязательный макрос. Требуется для проектов, в которых применяются файлы INX для создания для них файлов INF. Указывает, куда поместить выходной файл INF. В листинге 19.2 этот макрос использует значение, заданное макросу `NTTARGETFILES`, в результате чего INF-файл помещается в папку с другими выходными файлами.

- ◆ Макрос NTTARGETFILES.

Необязательный макрос. Указывает, что проект содержит файл `Makefile.inc`. Значение, присвоенное макросу, указывает имя выходного INF-файла, создаваемого утилитой Build из INX-файла проекта, и папку, в которую помещается этот файл.

## Исходные файлы, заголовки и библиотеки

Следующие макросы описывают исходные файлы, заголовки и библиотеки, применяемые для сборки выходных файлов.

◆ Макрос INCLUDES.

Необязательный макрос. Указывает нахождение папок, иных, чем папка проекта, и содержащих заголовочные файлы. Обычно в этих папках находятся заголовочные файлы для совместного использования в нескольких проектах. Пути папок типично указываются по отношению к папке проекта и разделяются точкой с запятой.

◆ Макрос SOURCES.

Обязательный макрос. Перечисляет исходные файлы проекта. По умолчанию файлы должны быть указаны в одной строчке. Обратная косая черта (\) — это символ продолжения строки, который позволяет перечислять файлы в отдельных строчках.

◆ Макрос TARGETLIBS.

Обязательный макрос. Указывает статические библиотеки, используемые драйвером. Драйверы UMDF должны привязываться, по крайней мере, к библиотеке Kernel32.lib. В примере в листинге 19.2 драйвер привязывается к трем библиотекам: Ntstrsafe.lib, Kernel32.lib и Advapi32.lib.

## Конфигурация сборки

Следующие макросы описывают конфигурацию сборки.

◆ Макрос C\_DEFINES.

Обязательный макрос для сборок, в которых применяется Unicode. Указывает переключатели периода компиляции, которые необходимо передать компилятору языка C. В примере в листинге 19.2 эта переменная указывает требуемый флаг для трассировки событий.

◆ Макрос DLLDEF.

Необязательный макрос. Указывает имя DEF-файла, который утилита Build передает библиотекарю, при сборке файлов экспорта и импорта. Если значение для этого макроса не указано, то утилита Build предполагает, что для DEF-файла используется имя, назначенное макросу TARGETNAME, и присваивает файлу имя \$(TARGETNAME).def.

◆ Макрос DLLENTRY.

Необязательный макрос. Указывает имя точки входа DLL как DllMain. Если значение этому макросу не назначено, то утилита Build использует имя по умолчанию DllMainCRTStartup.

◆ Макрос MSC\_WARNING\_LEVEL.

Необязательный макрос. Устанавливает уровень предупреждений компилятора. Рекомендуется устанавливать уровень предупреждений 4, что обеспечивает чистую сборку. При получении предупреждений, вызванных нестандартными возможностями, их можно отключить, что похоже на добавление следующей директивы в файл с кодом:

```
#pragma warning(disable:НомерПредупреждения)
```

Например, в листинге 19.2 второй экземпляр MSC\_WARNING\_LEVEL отключает предупреждение номер 4201.

◆ Макрос RUN\_WPP.

Необязательный макрос. Запускает препроцессор WPP. Если трассировка WPP не применяется, то эту директиву можно опустить. Опция -scan указывает, что файл Internal.h содержит определение функции сообщения трассировки. Тема трассировки WPP рассматривается в главе 11.

◆ Макрос USE\_MSVCRT.

Необязательный макрос. Указывает, что в проекте используется многопоточная библиотека языка С времени исполнения. Для свободных сборок применяется библиотека Msvcrt.lib, а для проверочных — библиотека Msvertd.lib.

◆ Макросы WIN32\_WINNT\_VERSION, \_NT\_TARGET\_VERSION и NTDDI\_VERSION.

Обязательные макросы. Обычно эти аспекты конфигурации сборки определяются средой сборки. Но сборка драйверов UMDF должна выполняться в среде сборки для Windows Vista. Указывая для этих трех макросов настройки, показанные в листинге 19.2, позволяет драйверам работать так же и под Windows XP.

## Файлы Makefile и Makefile.inc для драйвера Fx2\_Driver

Содержимое файла Makefile одинаковое для всех проектов драйверов, как обсуждалось в разд. "Вспомогательные файлы утилиты Build" ранее в этой главе. Проект драйвера Fx2\_Driver также содержит необязательный файл Makefile.inc. В этом файле записаны некоторые дополнительные директивы сборки для обработки выходного файла, не охватываемые файлом Makefile.def, которые служат для преобразования INX-файла проекта в соответствующий INF-файл. В листинге 19.3 приводится содержимое файла Makefile.inc для образца драйвера Fx2\_Driver.

### Листинг 19.3. Файл Makefile.inc для образца драйвера Fx2\_Driver

```
.SUFFIXES: .inx
STAMP=stampinf
.inx{$(OBJ_PATH)\$(0)}.inf:
    copy $(@B).inx $@
    $(STAMP) -f $@ -a $(BUILDARCH) -u
    $(UMDF_VERSION).$(UMDF_MINOR_VERSION).0
```

Значения директив этого файла таковы.

- ◆ Директива .SUFFIXES перечисляет расширения для сопоставления правил вывода. Распространенные расширения, такие как C, H и CPP, предопределены, но расширение INX необходимо указать в списке явно.
- ◆ Директива STAMP указывает командную строку, которая фактически создает файл INF из заданного файла INX. Стандартная переменная \$(OBJ\_PATH) указывает, что INF-файл необходимо поместить в папку для выходных файлов.
- ◆ В последней строке кода указывается версия UMDF и процессорная архитектура.

Для большинства драйверов UMDF можно использовать этот файл Makefile.inc с минимальными модификациями.

## Сборка драйвера Fx2\_Driver

Далее приводится процедура для сборки драйвера Fx2\_Driver для Windows Vista на платформе x86.

**Чтобы выполнить сборку драйвера Fx2\_Driver для Windows Vista на платформе x86:**

1. Откройте консольное окно Windows Vista and Windows Server Longhorn x86 Checked Build Environment в папке %wdk%.

2. С помощью команды `cd` перейдите в папку проекта `%wdk%\src\umdf\usb\Fx2_Driver\Final`.
3. Выполните следующую команду для сборки проекта:

```
build.exe -ceZ
```

Сборку проекта можно выполнить и другими способами, но приведенный пример показывает обычно применяемый набор флагов. Выходные файлы для драйвера `Fx2_Driver` помещаются в подпапку `objchk_wlh_x86\i386` папки проекта.

## Пример KMDF: сборка образца драйвера Osrusbf2

В этом разделе пошагово рассматривается создание проверочной сборки драйвера `Osrusbf2` для Windows Vista. Также в нем обсуждаются подробности некоторых вспомогательных файлов.

### Файл Sources для драйвера Osrusbf2

Файл `Sources` содержит большую часть информации, специфической для проекта, которая используется утилитой `Build` для сборки проекта. Он состоит из последовательности директив, которые назначают специфические для проекта значения набору макросов и переменных среды. Содержимое файла `Sources` для драйвера `Osrusbf2` (с удаленными комментариями) показано в листинге 19.4. Это типичный файл `Sources` для простого драйвера KMDF.

#### Листинг 19.4. Файл Sources для образца драйвера Osrusbf2

```
TARGETNAME=osrusbf2
TARGETTYPE=DRIVER
KMDF_VERSION=1
INF_NAME=osrusbf2
MISCFILES=$(OBJ_PATH)\$0\$(INF_NAME).inf
NTTARGETFILES=
INCLUDES=$(INCLUDES);..\inc
TARGETLIBS= $(DDK_LIB_PATH)\ntstrsafe.lib
SOURCES=driver.c \
    device.c \
    ioctl.c \
    bulkwr.c \
    Interrupt.c \
    osrusbf2.rc
ENABLE_EVENT_TRACING=1
!IFDEF ENABLE_EVENT_TRACING
C_DEFINES = $(C_DEFINES) -DEVENT_TRACING
RUN_WPP= $(SOURCES) \
    -km \
    -func:TraceEvents(LEVEL,FLAGS,MSD,...) \
    -gen:{km-WdfDefault.tpl}*.tmh
!ENDIF
```

В следующем разделе приводится краткое описание макросов и переменных среды, используемых в файле Sources для образца драйвера Osrusbf2. Многие из этих макросов и директив такие же, как и для драйверов UMDF. В таком случае дается сокращенное определение.

## Макросы файла Sources для драйвера Osrusbf2

В этом разделе рассматриваются макросы и переменные среды для образца драйвера Osrusbf2.

### Номер версии KMDF

Номер версии UMDF определяется макросом `KMDF_VERSION`. Это обязательный макрос. Указывает номер основной версии KMDF. В примере в листинге 19.4 применяется первая версия KMDF.

### Стандартные выходные файлы

Следующие макросы указывают стандартные выходные файлы сборки.

- ◆ Макрос `TARGETNAME`.

Обязательный макрос. Указывает имена для таких выходных файлов проекта, как файлы SYS и PDB. В листинге 19.4 применяется имя `osrusbf2`.

- ◆ Макрос `TARGETTYPE`.

Обязательный макрос. Указывает тип выходного двоичного файла. В листинге 19.4 значение `DRIVER` означает, что необходимо выполнить сборку драйвера режима ядра.

### Специальные выходные файлы

Следующие макросы описывают специальные выходные файлы, такие как INF-файл, создаваемый утилитой Build при обработке INX-файла.

- ◆ Макрос `INF_NAME`.

Необязательный макрос. Указывает имя выходного INF-файла, создаваемого утилитой Build из INX-файла проекта. В листинге 19.4 применяется значение `Osrusbf2.inf`.

- ◆ Макрос `MISCFILES`.

Необязательный макрос. Указывает папку для помещения созданного INF-файла. В листинге 19.4 файл INF помещается в папку с другими выходными файлами.

- ◆ Макрос `NTTARGETFILES`.

Необязательный макрос. Указывает, что проект содержит файл `Makefile.inc`. В листинге 19.4 значение не указано. Вместо этого папка для выходных файлов указывается в макросе `INF_NAME`.

### Исходные файлы, заголовки и библиотеки

Следующие макросы описывают исходные файлы, заголовки и библиотеки, применяемые для сборки выходных файлов:

- ◆ Макрос `INCLUDES`.

Необязательный макрос. Указывает папки, содержащие разделяемые заголовочные файлы.

◆ Макрос SOURCES.

Обязательный макрос. Перечисляет исходные файлы проекта. По умолчанию файлы должны быть указаны в одной строчке. Обратная косая черта (\) — это символ продолжения строки, который позволяет перечислять файлы в отдельных строчках.

◆ Макрос TARGETLIBS.

Необязательный макрос. Указывает библиотеки, которые нужно использовать. Для драйверов KMDF не требуется никаких LIB-файлов, но в данном проекте применяется библиотека безопасных строк Ntstrsafe.lib.

## Конфигурация сборки

Следующие макросы описывают конфигурацию сборки.

◆ Макрос C\_DEFINES.

Обязательный макрос для сборок, в которых применяется Unicode. Указывает переключатели периода компиляции, которые необходимо передать компилятору. Для драйвера Osrusbfx2 макрос указывает требуемый флаг для трассировки событий.

◆ Макрос ENABLE\_EVENT\_TRACING.

Необязательный макрос. Специальный макрос (определяемый в образце драйвера), который разрешает трассировку событий WPP. Чтобы запретить трассировку в данном образце, удалите или закомментируйте соответствующую строку. Тема трассировки WPP рассматривается в главе 11.

◆ Макрос FDEF/!ENDIF.

Необязательный макрос. Работает по большому счету таким же образом, как и его эквивалент в языке С. Код между макросами исполняется только в случае удовлетворения условия. В примере в листинге 19.4 две директивы между этими макросами исполняются, только если разрешена трассировка WPP.

◆ Макрос RUN\_WPP.

Необязательный макрос. Запускает препроцессор WPP. Опция gen указывает файл шаблона WDF, который поддерживает сотни трассировочных сообщений, специфичных для WDF.

## Файлы Makefile и Makefile.inc для драйвера Osrusbfx2

Содержимое файла Makefile, одинаковое для всех проектов драйверов, как обсуждалось в разд. "Вспомогательные файлы утилиты Build" ранее в этой главе. Проект драйвера Osrusbfx2 также содержит необязательный файл Makefile.inc. В этом файле записаны некоторые дополнительные директивы сборки для обработки выходного файла, не охватываемые файлом Makefile.def, которые служат для преобразования INX-файла проекта в соответствующий файл INF. В листинге 19.5 приводится содержимое файла Makefile.inc для образца драйвера Osrusbfx2.

**Листинг 19.5. Файл Makefile.inc для образца драйвера Osrusbfx2**

```
_LNG=$ (LANGUAGE)
_INX=.
STAMP=stampinf -f $@ -a ${_BUILDARCH}
```

```
$ (OBJ_PATH)\$0\$(INF_NAME).inf: $(INX)\$(INF_NAME).inx
copy $(INX)\$(@B).inx $@
$(STAMP)
```

Значение директив в листинге 19.5 следующее:

- ◆ директива `_LNG` указывает язык сборки. Значение WDK по умолчанию — американский английский;
- ◆ директива `_INX` указывает, что INX-файл находится в папке проекта;
- ◆ директива `STAMP` указывает командную строку, которая фактически создает файл INF из заданного файла INX;
- ◆ директива `OBJ_PATH` указывает папку, в которую нужно поместить выходной файл.

Драйверы KMDF могут поддерживать интерфейс WMI, для чего требуются дополнительные директивы в файле Makefile.inc. В листинге 19.6 приводится содержимое файла Makefile.inc для образца драйвера KMDF Featured Toaster. Первая половина этого листинга предназначена для файла INX и идентична листингу 19.5. Вторая же половина содержит директивы для создания файла ресурса MOF проекта.

#### Листинг 19.6. Файл Makefile.inc для образца драйвера KMDF Featured Toaster

```
_LNG=$(LANGUAGE)
_INX=.
STAMP=stampinf -f $@ -a $(BUILDARCH)
$(OBJ_PATH)\$(0)\$(INF_NAME).inf: $(INX)\$(INF_NAME).inx
copy $(INX)\$(@B).inx $@
$(STAMP)
mofcomp: $(OBJ_PATH)\$(0)\toaster.bmf
$(OBJ_PATH)\$(0)\toaster.bmf: toaster.mof
mofcomp -WMI -B:$(OBJ_PATH)\$0\toaster.bmf toaster.mof
wmimofck -m -h$(OBJ_PATH)\$0\ToasterMof.h -w$(OBJ_PATH)\$0\htm
$(OBJ_PATH)\$(0)\toaste r.bmf
```

Для большинства драйверов KMDF эти примеры можно использовать с минимальными модификациями, которые, в основном, состоят в изменении имен файлов в разделе WMI. Для драйверов, не поддерживающих WMI и не использующих INX-файл, файл Makefile.inc не требуется.

## Сборка драйвера Osrusbf2

Сборка драйвера Osrusbf2 выполняется следующим образом.

### Чтобы выполнить сборку драйвера Osrusbf2:

1. Откройте консольное окно **Windows Vista and Windows Server Longhorn x86 Checked Build Environment** в папке %wdk%.
2. С помощью команды `cd` перейдите в папку %wdk%\src\kmdf\osrusbf2\sys\final.
3. Выполните следующую команду для сборки проекта:

```
build.exe -ceZ
```

Сборку проекта можно выполнить и другими способами, но приведенный пример показывает наиболее употребляемый набор флагов.

Выходные файлы сборки помещаются в подпапку objchk\_wlh\_x86\i386 проекта.

# ГЛАВА 20

## Установка драйверов WDF

Прежде чем драйвер можно использовать (или только для его тестирования и отладки, или уже для управления устройством), его нужно установить. Процедуры для установки драйвера существенно отличаются от процедур для установки приложений.

В этой главе рассматривается создание инсталляционного пакета драйвера WDF, включая подписание драйверов, и сам процесс установки.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b> Библиотеки DLL соинсталляторов Файл Stampinf.exe	%wdk%\redist\wdf %wdk%\bin\<amd64   ia64   i386>
<b>Образцы драйверов</b> Файл INF драйвера Fx2_Driver Файл INF драйвера Osrusbfx2	%wdk%\build\src\umdf\usb\fx2_driver %wdk%\build\src\kmdf\osrusbfx2
<b>Документация WDK</b> Раздел "Device Installation Design Guide" <sup>1</sup> Раздел "Installing a Framework-based Driver" <sup>2</sup> Раздел "Installing UMDF Drivers" <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79351">http://go.microsoft.com/fwlink/?LinkId=79351</a> <a href="http://go.microsoft.com/fwlink/?LinkId=79352">http://go.microsoft.com/fwlink/?LinkId=79352</a> <a href="http://go.microsoft.com/fwlink/?LinkId=79345">http://go.microsoft.com/fwlink/?LinkId=79345</a>
<b>Прочее</b> Различные доклады на тему установки драйверов на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=79354">http://go.microsoft.com/fwlink/?LinkId=79354</a>

<sup>1</sup> Руководство по разработке драйверных инсталляций. — Пер.

<sup>2</sup> Установка драйверов WDF. — Пер.

<sup>3</sup> Установка драйверов UMDF. — Пер.

## Основы установки драйверов

Разработчики драйверов должны различать установку драйверов для двух разных целей.

- ◆ Установка драйвера на испытательную систему для тестирования.

На протяжении цикла разработки драйвера такую установку приходится выполнять многократно, чтобы проверить работу драйвера на разных этапах разработки.

- ◆ Окончательная установка на систему пользователя для штатной эксплуатации драйвера.

Для обновления драйверов применяются инструменты и процедуры, подобные тем, которые применяются для установки новых драйверов. Если не оговорено иное, то в этой главе термин "установка" означает установку драйверов как для тестирования, так и для эксплуатации.

### Ключевые задачи по установке драйверов

При установке драйверов необходимо выполнить следующие ключевые задачи:

- ◆ скопировать двоичный файл драйвера в соответствующую папку на компьютере;
- ◆ зарегистрировать драйвер в Windows и указать, каким образом драйвер должен загружаться;
- ◆ добавить необходимую информацию в реестр;
- ◆ скопировать или установить все связанные компоненты, такие как вспомогательные DLL или приложения.

Чтобы выполнить эти задачи, необходимо собрать требуемые файлы в драйверный пакет и точно определить, каким образом необходимо выполнить каждую ключевую задачу.

### Инструменты и методы установки

Установку драйвера можно выполнить несколькими разными способами.

- ◆ **Предоставить выполнение установки менеджеру PnP.** Это основной способ установки пользователями драйверов для новых устройств. Пользователь просто вставляет устройство в разъем, и менеджер PnP выдает инструкции по дальнейшим действиям. (Чтобы обеспечить правильное выполнение установки менеджером PnP при первоначальном подключении устройства к компьютеру, в процессе разработки драйверный пакет необходимо подвергнуть всестороннему тестированию.) Но этот подход применим только к новым устройствам. Для обновления уже установленных драйверов разработчики должны применять другие подходы.

Информацию по установке драйверов с помощью менеджера PnP см. в разд. "Установка драйверов с помощью менеджера PnP" далее в этой главе.

- ◆ **Применить специальное установочное приложение.** Для этого можно применять два свободно распространяемых настраиваемых приложения, поставляемые с WDK, — Driver Package Installer (DPIInst) и Driver Install Frameworks for Applications (DIFxApp). Для установки драйверов, требующих выполнения специальных процедур, можно реализовать свое специальное установочное приложение.

Дополнительную информацию по установочным приложениям см. в разд. "Установка драйверов с помощью DPIInst или DIFxApp" далее в этой главе.

- ◆ **С помощью Диспетчера устройств Windows.** Этот инструмент можно использовать как для установки, так и для обновления драйверов, но обычно он применяется только разработчиками или опытными пользователями.  
Информацию по применению этого средства см. в разд. "Обновление драйверов с помощью Диспетчера устройств" далее в этой главе.
- ◆ **Воспользоваться инструментом DevCon из набора WDK.** Разработчики могут использовать этот инструмент командной строки для управления драйверами, включая установку и обновление. Он часто применяется для установки драйверов на тестовые системы.  
Информацию по применению этого средства см. в разд. "Установка и обновление драйверов с помощью инструмента DevCon" далее в этой главе.

Процедуры для установки драйверов не являются специфичными для WDF и в этой книге подробно не рассматриваются. Дополнительную информацию по теме также см. в разделе **Device Installation** (Установка устройств) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79294>.

## Аспекты, принимаемые во внимание при установке драйверов WDF

Основы подготовки к установке драйверов WDF и сама установка по существу такие же, как для других драйверных моделей Windows. Основная разница заключается в том, что драйверы WDF зависят от определенной версии библиотеки инфраструктуры. Текущая версия инфраструктуры, установленная на целевой системе, может быть устаревшей или же, в случае с системами более ранними, чем Windows Vista, инфраструктура может отсутствовать вообще.

Поэтому процедура установки драйвера WDF должна убедиться в наличии установленной на системе необходимой версии инфраструктуры. В этом разделе рассматривается управление версиями WDF и применение соинсталляторов WDF для проверки наличия правильной версии инфраструктуры.

## Управление версиями WDF и установка драйверов

С помощью управления версиями WDF драйверы, привязанные к разным основным версиям UMDF или KMDF, можно устанавливать и запускать параллельно на одной и той же системе. Драйверы WDF привязываются к основной версии инфраструктуры, под которую они скомпилированы, и игнорируют все остальные основные версии, которые могут быть установленными в системе. Это позволяет всегда обеспечить привязку драйвера к основной версии инфраструктуры, под которую он был разработан и протестирован.

Для каждой основной версии может существовать несколько дополнительных версий. Кроме исправления ошибок, каждая дополнительная версия может предоставлять дополнительные возможности или методы интерфейса DDI. Но с целью обеспечения обратной совместимости, при установке новой дополнительной версии, существующие методы интерфейса DDI или возможности не меняются. Это означает, что драйвер WDF должен исполняться без проблем на любой дополнительной версии соответствующей основной версии, при условии, что номер дополнительной версии, установленной на компьютере, такой же или больше, чем номер дополнительной версии, под какую драйвер был разработан.

### Примечание

Соинсталлятор KMDF не добавляет инфраструктуру в список установленных приложений в служебном приложении **Programs** в Панели управления (которое в более ранних версиях Windows называлось **Add or Remove Programs** (Установка и удаление программ)). Таким образом, обеспечивается, что пользователь случайно не удалит инфраструктуру.

## Обновления дополнительных версий

На одном компьютере нельзя установить две дополнительные версии одной и той же основной версии инфраструктуры. При установке новой дополнительной версии старая дополнительная версия автоматически удаляется. Драйверы, использующие старую дополнительную версию, автоматически привязываются к новой дополнительной версии.

Обновления дополнительных версий могут содержать новые возможности и добавления к DDL. Но т. к. новые дополнительные версии гарантированно обратно-совместимые, то драйверам WDF не нужно ничего делать для способствования установки новой дополнительной версии. Единственное, что может потребоваться, — так это перезагрузить систему.

Для KMDF, если старая дополнительная версия уже загружена, когда устанавливается новая дополнительная версия, перезагрузка системы обязательна.

Для UMDF, если старая дополнительная версия уже загружена, когда устанавливается новая дополнительная версия, соинсталлятор пытается остановить наиболее организованным образом все стеки устройств, привязанные к старой дополнительной версии. Если затронутые стеки устройств поддерживают запросы на удаление по результатам опроса и запросы на удаление, соинсталлятор может перевести стеки в состояние `Stopped` и обновить дополнительную версию без перезагрузки системы. Но если один или несколько стеков устройств не поддерживают удаление по результатам опроса и запросы на удаление или если в приложении имеется открытый дескриптор какого-либо устройства, то перезагрузка системы обязательна.

Независимо от того, требуется перезагрузка или нет, при перезагрузке драйверов все драйверы WDF, которые были привязаны к старой дополнительной версии, должны привязаться к новой дополнительной версии.

## Обновления основных версий

При установке новой основной версии WDF существующие драйверы продолжают поддерживать привязку к предыдущей основной версии. Но вам следует обновить ваши драйверы под самую последнюю установленную основную версию WDF, т. к. она содержит все последние исправления ошибок и новые возможности. Обратите внимание, что для новой основной версии инфраструктуры не гарантируется обратная совместимость с предыдущими основными версиями. Всегда выполняйте регрессивное тестирование драйверов, чтобы быть уверенным в том, что они работают правильно с новой основной версией.

## Распространение инфраструктуры

Корпорация Microsoft распространяет инфраструктуры WDF несколькими способами.

- ◆ Windows Vista и более поздние версии Windows содержат собственные версии KMDF и UMDF.
- ◆ Набор WDK содержит соинсталляторы WDF, которые поставщики могут включать в свои драйверные пакеты.

Эти соинсталляторы устанавливают инфраструктуру на целевые системы при установке устройства, если на системе еще не установлена самая последняя версия инфраструктуры.

- ◆ Обычно новая основная версия инфраструктуры устанавливается, когда пользователь устанавливает драйвер, для которого требуется эта версия, но она также может распространяться посредством сервисных пакетов Windows.

Если обновление необходимо, чтобы исправить критическую ошибку безопасности, то служба обновлений Windows (Windows Update) автоматически загружает новую дополнительную версию инфраструктуры на компьютер пользователя. В противном случае новая дополнительная версия устанавливается, только если пользователь устанавливает драйвер, для которого она требуется.

Дополнительную информацию по теме см. в разделе **Framework Library Versions** (Версии библиотеки инфраструктуры) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79355>.

## Привязка драйверов к инфраструктуре

Драйверы WDF динамически привязываются к инфраструктуре во время процесса загрузки. В этом разделе описывается процесс привязки как для драйверов KMDF, так и для драйверов UMDF.

### Драйверы KMDF

При сборке драйверы KMDF связываются статически с библиотекой `WdfDriverEntry.lib`. Эта библиотека содержит информацию о версии KMDF в статической структуре данных, которая становится частью двоичного файла драйвера. Внутренняя функция в этой библиотеке инкапсулирует процедуру `DriverEntry` драйвера. При загрузке драйвера эта внутренняя функция становится точкой входа драйвера. При загрузке драйвера происходит следующее:

1. Внутренняя функция вызывает загрузчик KMDF и передает номер версии библиотеки KMDF, к которой необходимо выполнить привязку.
2. Загрузчик выполняет проверку, была ли указанная основная версия библиотеки инфраструктуры уже загружена. Если указанная основная версия еще не была загружена, то загрузчик запускает среду исполнения KMDF.
3. Если среда исполнения запускается успешно, то загрузчик добавляет драйвер в качестве клиента среды исполнения и возвращает релевантную информацию внутренней функции.

Если драйверу требуется более новая версия библиотеки времени исполнения, чем версия, загруженная в данное время, загрузка завершается неудачей, о чем делается соответствующая запись в системном журнале событий.

Драйверами BSD (boot-start drivers) называются драйверы, которые устанавливаются во время загрузки операционной системы. Сценарий загрузки BSD-драйверов KMDF несколько отличается от только что описанного сценария, т. к. перед загрузкой драйвера сначала должна быть загружена среда исполнения KMDF. При установке соинсталлятор получает информацию из INF-файла или реестра, чтобы определить, является ли драйвер драйвером BSD. Если является, то тогда соинсталлятор выполняет следующие действия:

- ◆ изменяет тип запуска среды исполнения KMDF, чтобы загрузчик Windows запускал ее при загрузке операционной системы;

- ◆ устанавливает такой порядок загрузки, что среда исполнения KMDF загружается перед загрузкой драйвера клиента.

## Драйверы UMDF

Все драйверы UMDF должны содержать стандартный заголовочный файл wudfddi.h, экспортирующий информацию, в которой указывается версия UMDF, требуемая драйвером. При загрузке драйвера среда исполнения UMDF проверяет экспортированное значение, чтобы удостовериться в том, что драйверу требуется более ранняя или такая же версия инфраструктуры, как и установленная в настоящее время. Если драйверу требуется более новая версия, чем установленная, то загрузчик завершает работу неудачей.

## Пакеты соинсталляторов WDF

Пакеты соинсталляторов WDF представляют собой библиотеки DLL, которые разработчики могут распространять со своими продуктами согласно условиям лицензионного договора, представленным при установке поставщиком пакета разработки WDK. Для каждого поддерживаемого типа центрального процессора существует отдельный набор соинсталляторов. А именно:

- ◆ для UMDF и KMDF применяются отдельные соинсталляторы, которые находятся в папке %wdk%\redist\wdf. Они рассматриваются более подробно в следующих разделах;
- ◆ набор разработчика WDK также содержит соинсталлятор для компонента WinUSBCoInstaller.dll архитектуры WinUSB, который предоставляет поддержку USB-драйверам UMDF. Библиотека DLL соинсталлятора WinUSB находится в папке %wdk%\redist\winusb.

Драйверные пакеты должны содержать соответствующие соинсталляторы WDF. Номера версий этих соинсталляторов должны быть большими или такими же, как и номер версии WDF, для которой драйвер был скомпилирован.

Набор разработчика WDK содержит как проверочные, так и свободные версии соинсталляторов.

- ◆ Свободные версии соинсталляторов применяются при установке драйверов WDF на свободные сборки Windows, такие как поставляемые версии Windows, установленные на компьютерах большинства пользователей.
- ◆ Проверочные сборки соинсталляторов (которые обозначаются префиксом \_chk перед именем свободной сборки) применяются только для тестирования драйверов для установки их на проверочные или частично проверочные версии Windows. Частично проверочная версия Windows — это свободная версия, в которой первоначальные компоненты HAL и ntoskrnl.exe заменены их проверочными версиями.

Информацию о создании частично проверочной сборки Windows см. в разделе **Installing Just the Checked Operating System and HAL** (Установка проверочных версий только операционной системы и HAL) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79774>.

Независимо от того, в какой среде была выполнена сборка драйвера — в проверочной или свободной, — оба типа соинсталляторов работают с обоими типами драйверов.

### Внимание!

Тип сборки соинсталлятора должен быть таким же, как и версия Windows, на которой будет установлен драйвер. Проверочную версию соинсталлятора нельзя использовать для установки драйвера на свободную сборку Windows и наоборот.

### Пакет соинсталлятора UMDF

Свободно распространяемый пакет соинсталлятора UMDF называется `WudfUpdate_MMmm.dll`, где:

- ◆ *MM* указывает номер основной версии UMDF;
- ◆ *mmm* указывает номер дополнительной версии UMDF.

Например, DLL соинсталлятора для UMDF версии 1.5 называется `WUDFUpdate_01005.dll`

Так как UMDF содержит как компоненты пользовательского режима, так и компоненты режима ядра, соинсталлятор содержит смесь файлов SYS, DLL и EXE. Назначение этих файлов описывается в табл. 20.1.

**Таблица 20.1. Компоненты соинсталлятора UMDF**

Компонент	Установлен в папку	Описание
<code>Wudfrd.sys</code>	<code>%Windir%\System32\Drivers</code>	Отражатель
<code>Wudfhost.exe</code>	<code>%Windir%\System32</code>	Хост драйвера
<code>Wudfsvc.dll</code>	<code>%Windir%\System32</code>	Менеджер драйверов
<code>WUDFPlatform.dll</code>	<code>%Windir%\System32</code>	Компонент пользовательского режима среды исполнения UMDF
<code>Wudfpf.sys</code>	<code>%Windir%\System32\Drivers</code>	Компонент режима ядра среды исполнения UMDF
<code>Wudfx.dll</code>	<code>%Windir%\System32</code>	Библиотека инфраструктуры UMDF
<code>Wudfcoinstaller.dll</code>	<code>%Windir%\System32</code>	Соинсталлятор конфигурирования UMDF, который занимается конфигурированием устройств и драйверов UMDF

Компоненты, перечисленные в табл. 20.1, и их место в архитектуре UMDF рассматриваются в главе 4.

Рефлектор может быть установлен как сервис, верхний драйвер фильтра или нижний драйвер фильтра. Как именно, определяется следующим образом:

- ◆ если драйвер UMDF является функциональным драйвером или если стек пользовательского режима содержит функциональный драйвер, тогда рефлектор нужно установить как сервис;
- ◆ если драйвер UMDF является драйвером фильтра или если стек пользовательского режима содержит только драйверы фильтра, тогда рефлектор нужно установить как самый высший драйвер фильтра высокого уровня в стеке режима ядра;
- ◆ если устройство не ассоциировано с сервисом, т. е. устройство является "сырым" устройством, тогда рефлектор нужно установить как самый высший драйвер фильтра низкого уровня.

Дополнительную информацию по теме см. в разделе **Adding the Reflector** (Установка рефлектора) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80058>.

## Пакет соинсталлятора KMDF

Свободно распространяемый пакет соинсталлятора KMDF называется `WdfCoInstallerMMmm.dll`, где *MM* и *mm* означают основную и дополнительную версии соответственно.

Например, DLL соинсталлятора для KMDF версии 1.5 называется `WdfCoInstaller01005.dll`. Библиотека DLL содержит два SYS-файла, которые устанавливаются в папку `%Windir%\System32\Drivers`. Описание этих файлов приводится в табл. 20.2.

**Таблица 20.2. Компоненты соинсталлятора KMDF**

Компонент	Установлен в папку	Описание
<code>WdfLdr.sys</code>	<code>%Windir%\System32\Drivers</code>	Загружает среду исполнения инфраструктуры
<code>WdfMM000.sys</code>	<code>%Windir%\System32\Drivers</code>	Собственно среда исполнения инфраструктуры, где <i>MM</i> означает номер основной версии

## Компоненты драйверного пакета WDF

Чтобы подготовить драйвер к установке, необходимо создать пакет, содержащий все компоненты, необходимые для выполнения установки. Ключевым компонентом пакета является INF-файл, который содержит набор данных и инструкций для целевой системы. Независимо от способа установки драйвера, основные элементы пакета по существу одинаковые. Таким образом, установка драйвера сводится к сборке пакета и реализации INF-файла для этого пакета.

Поставляемый драйверный пакет WDF содержит следующие компоненты.

- ◆ **Двоичные файлы драйвера.** В зависимости от типа драйвера, UMDF или KMDF, пакет, как минимум, должен содержать DLL- или SYS-файл драйвера соответственно. Сборка двоичного файла драйвера должна быть выполнена для версии инфраструктуры, указанной в файле INF, но также может быть и для более новой дополнительной версии. Пакет может также содержать разные необязательные двоичные файлы, такие как вспомогательные библиотеки DLL, поставщики страницы свойств и родственные приложения.
- ◆ **Один или оба соинсталлятора WDF.** Для драйверов UMDF и KMDF пакет должен содержать их соответствующие соинсталляторы. Но для некоторых драйверов в пакет необходимо включать оба соинсталлятора:
  - для USB-драйверов UMDF оба соинсталлятора требуются потому, что эти драйверы зависят от предоставляемого системой компонента WinUSB, который основан на KMDF. Такой пакет также должен содержать соинсталлятор WinUSB;
  - гибридные драйверы содержат как компоненты UMDF, так и компоненты KMDF, поэтому для них необходимы соинсталляторы для обеих инфраструктур.
- ◆ **Файлы INF.** Файл INF является центральным компонентом установочного пакета. Он представляет собой текстовый файл, содержащий директивы, указывающие Windows, как устанавливать драйвер. Обычно пакет содержит или один файл INF для всего пакета, или отдельные файлы INF для каждой процессорной архитектуры.

Файл INF для драйвера WDF содержит следующие данные:

- общую информацию об устройстве, например, производитель устройства, класс установки и номер версии;
- имена файлов, их местонахождение на диске дистрибутива и куда они должны быть установлены на систему пользователя;
- директивы для создания или модификации элементов реестра для драйвера или устройства;
- установочные директивы относительно того, какие драйверы нужно установить, какие двоичные файлы содержат драйвер и список драйверов, которые нужно загрузить для устройства;
- директивы для установки специфичной для WDFF конфигурационной информации.

◆ **Файл каталога (с расширением CAT), подписанный цифровой подписью.** Подписанный цифровой подписью файл каталога играет роль сигнатуры для файлов пакета. Файл каталога требуется для 64-битных версий Windows Vista и более поздних версий Windows и настоятельно рекомендуется для всех драйверов.

Цифровая подпись пакета упрощает процесс установки и предоставляет пользователю два очень важных дополнительных преимущества:

- с помощью подписи пользователи могут определить происхождение пакета;
- пользователи могут использовать подпись, чтобы проверить, не подвергался ли пакет несанкционированным модификациям после того, как он был подписан. Например, с помощью цифровой подписи можно убедиться в том, что драйвер не был модифицирован или заражен вирусом.

Дополнительную информацию по этому вопросу см. в разд. "Подписание и распространение драйверного пакета" далее в этой главе.

◆ **Необязательные компоненты.** Пакет может содержать необязательные компоненты, такие как сопутствующие приложения, вспомогательные библиотеки DLL или пиктограммы.

## Создания INF-файла для драйверного пакета WDF

В формате INF каждая строка может быть одним из двух базовых компонентов.

- ◆ **Раздел.** Файл INF состоит из нескольких разделов, которые обозначаются именем, заключенным в квадратные скобки, например [Version]. Некоторые стандартные разделы являются обязательными для всех INF-файлов, а другие нет. Можно также определять свои разделы, специфические для определенного INF-файла.
- ◆ **Директива.** Каждый раздел содержит один или несколько элементов формата **ключ=значение**, называющихся директивами, в которых указываются данные для установки.

В листинге 20.1 приводится раздел [Version] INF-файла образца драйвера Fx2\_Driver. Раздел содержит шесть директив.

**Листинг 20.1. Раздел [Version] образца драйвера Fx2\_Driver**

```
[Version]
Signature="$Windows NT$"
Class=Sample
ClassGuid={78A1C341-4539-11d3-B88D-OOC04FAD5171}
Provider=%MSFTUMDF%
DriverVer=10/13/2006, 6.0.5753.0
CatalogFile=wudf.cat
```

Директива DriverVer указывает дату и версию сборки драйвера. Сборка данного драйвера была выполнена 13 октября 2006 г.

Большинство разделов и директив в них для драйверов WDF по существу такие же, как и для подобных драйверов WDM. Например, около двух третьих содержимого с начала INF-файла образца драйвера Osrusbfx2 почти идентично содержимому INF-файла для версии WDM-драйвера. Основная разница между INF-файлами для драйверов WDF и WDM заключается в том, что первые должны содержать набор разделов для обслуживания соинсталляторов WDF.

В этом разделе в основном рассматриваются специфические для WDF части INF-файла.

Подробную информацию по этому вопросу см. в разделе **Creating an INF File** (Создание файлов INF) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79356>.

## **Широко применяемые разделы INF-файла**

Разделы любого конкретного INF-файла и их содержимое зависят от драйвера, для которого создается данный файл. Наиболее широко применяются следующие разделы.

- ◆ [Version]

Кроме версии драйвера, в этом разделе также может указываться различная основная информация о драйвере, например имя файла каталогов пакета.

- ◆ [Manufacturer]

Этот раздел содержит информацию о производителе устройства или устройств, которые могут быть установлены с помощью данного INF-файла.

- ◆ [SourceDisksNames] и [SourceDisksFiles]

В этих двух разделах указывается местонахождение файлов, которые нужно установить.

- ◆ [DestinationDirs]

Этот раздел предоставляет информацию о папках, в которые нужно установить файлы.

- ◆ [ClassInstall32]

Этот раздел устанавливает новый класс для установки устройств.

- ◆ [Strings]

Этот раздел содержит заполнители и их соответствующие строки. Заполнители применяются в других местах INF-файла и подменяются их соответствующими строками. Этот раздел особенно полезен при локализации.

Несколько разделов предназначены для работы с соинсталляторами WDF. Эти разделы рассматриваются в разд. "Файлы INF драйверов WDF: разделы соинсталляторов" далее в этой главе.

## Инструменты для работы с INF-файлами

В наборе разработчика WDK не предоставляется инструментов для автоматического создания INF-файлов — эти файлы необходимо создавать вручную в обычном текстовом редакторе. Но создания всего INF-файла обычно можно избежать. Просто скопируйте содержимое INF-файла для походящего образца драйвера, а потом модифицируйте его под нужды вашего драйвера.

### Полезная информация

Структуру и синтаксис INF-файла можно проверить с помощью инструмента набора WDK — ChkINF, который находится в папке %wdk%\tools\chkinf. Документацию по использованию этого инструмента можно найти в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79776>. Но обратите внимание на то, что в настоящее время с помощью этого инструмента нельзя проверить разделы WDF файла INF.

## Файлы INF для разных процессорных архитектур

Драйверы обычно необходимо предоставлять для всех поддерживаемых типов процессорных архитектур. Но для каждой процессорной архитектуры требуется отдельная сборка драйвера. Файлы, созданные утилитой Build для разных сборок, обычно имеют одинаковые имена, поэтому разные сборки хранятся в отдельных целевых папках. Некоторые аспекты установки драйвера зависят от архитектуры процессора системы, поэтому INF-файлы должны содержать все данные, необходимые для поддержки всех трех архитектур.

Файлы INF для поддержки разных процессорных архитектур можно создавать двумя основными способами.

- ◆ **Поместить данные для разных архитектур в отдельные разделы одного файла INF.** Этот метод удобен тем, что все данные можно поместить в один файл, но его недостатком является сравнительно большой и сложный INF-файл.
- ◆ **Создать отдельный INF-файл для каждой архитектуры.** Эти файлы получаются меньшего объема и менее сложными, но многие данные в них, такие как, например, версия KMDF, не зависят ни от какой конкретной процессорной архитектуры, и идентичны для всех трех файлов. Это означает, что при любом изменении разделяемых данных, их необходимо обновить во всех трех файлах отдельно.

Наиболее эффективный подход, который применяется в образцах драйверов WDF, заключается в совместном использовании обоих методов посредством применения в проекте INX-файла. Файл INX является независимой от архитектуры версией файла INF. Почти все содержимое файла INX идентично содержимому файла INF, но в файле INX вместо зависимых от процессорной архитектуры значений ставятся заполнители. Для драйверов UMDF заполнителями в файле INX обозначаются версия соинсталлятора и версия библиотеки UMDF.

При сборке проекта утилите Build дается указание применить инструмент Stampinf, который заменяет заполнители в выходном файле INF соответствующими значениями для конкретной процессорной архитектуры и помещает его в папку вместе с двоичными файлами драйвера. В случае изменения каких-либо данных, эти изменения вносятся в файл INX, и выполняется повторная сборка проекта, чтобы обновить файлы INF.

В главе 19 рассматривается, как организовать проект для использования файла INX.

Для примера, в листинге 20.2 приводится раздел [Manufacturer] файла INX для образца драйвера Osrusbf2. В этом файле для драйверов независимых поставщиков слово "Microsoft" было бы заменено соответствующим именем поставщика.

**Листинг 20.2. Раздел [Manufacturer] файла INX драйвера Osrusbf2 и соответствующие разделы файлов INF**

Раздел файла INX.

[Manufacturer]

%MfgName%=Microsoft, NT\$ARCH\$

В соответствующем файле INF для архитектуры x86 этот раздел будет выглядеть таким образом:

[Manufacturer]

%MfgName%=Microsoft, NTx86

В соответствующем файле INF для архитектуры x64 этот раздел будет выглядеть таким образом:

[Manufacturer]

%MfgName%=Microsoft, NTAMD64

**Внимание!**

Проекты, в которых используется INX-файл, должны также содержать файл Makefile.inc, в котором утилите Build указывается, каким образом исполнять утилиту Stampinf.exe для создания INF-файлов. Подробную информацию по этому вопросу смотрите в главе 19.

## Файлы INF драйверов WDF: разделы соинсталляторов

Основное различие между INF-файлами для драйверов более ранних драйверных моделей и для драйверов модели WDF заключается в том, что INF-файл для последних должен содержать данные и инструкции для выполнения следующих задач:

- ◆ копирования соинсталлятора на целевой компьютер;
- ◆ указания этого соинсталлятора как соинсталлятора устройства.

Элементы в разделе [Manufacturer] указывают имя раздела (обычно это будет имя производителя) и процессорную архитектуру. В разделе [Manufacturer] драйвера Osrusbf2 имя раздела указывается как [Microsoft]. Этот раздел содержит список идентификаторов аппаратного обеспечения, каждый из которых указывает значение *DDInstall*, которое нужно использовать для установки устройства, соответствующего данному идентификатору. Например, раздел [Microsoft] для сборки под процессорную архитектуру x86 файла INF драйвера Osrusbf2 содержит один идентификатор аппаратного обеспечения со значением *DDInstall* равным Osrusbf2.dev (листинг 20.3).

**Листинг 20.3. Раздел [Microsoft] файла INF образца драйвера Osrusbf2**

```
[Microsoft.NTx86]
%USB\VID_045E&PID_930A.DeviceDesc%=Osrusbf2.dev, USB\VID_0547&PID_1002
%Switch.DeviceDesc%=Switch.Dev,
{6FDE7521-1B65-48ae-B628-80BE62016026}\OsrUsbFxRawPdo
```

Раздел первичного соинсталлятора WDF называется [*DDInstall.Coinstallers*] и содержит директивы *CopyFiles* и *AddReg* для установки соинсталлятора и ассоциирования его с устройством.

### Примечание

Части имен раздела, выделенные жирным шрифтом, являются обязательными. А части имен раздела, представленные курсивом, могут иметь любое определенное пользователем значение.

Соинсталлятор, в свою очередь, исполняет раздел [*DDInstall.Wdf*], который содержит директивы по установке для соинсталлятора. Обычно несколько других сопутствующих разделов, таких как раздел *DestinationDirs*, указывают, куда нужно копировать файлы.

Для драйверов KMDF раздел [*DDInstall.Wdf*] содержит одну директиву, *KmdfService*, как показано в следующем примере:

```
KmdfService = DHverService, Wdf-install
```

Директива *KmdfService* назначает имя сервису режима ядра драйвера и указывает на раздел [*Wdf-install*], который содержит директиву *KmdfLibraryVersion*, как показано в следующем примере:

```
KmdfLibraryVersion = WdfLibraryVersion
```

Значение *WdfLibraryVersion* — это номер, например 1.5 или 2.2, который указывает минимальные основную и дополнительную версии библиотеки KMDF, требуемые драйвером.

Для драйверов UMDF раздел [*DDInstall.Wdf*] содержит, по крайней мере, две директивы. В табл. 20.3 приводятся некоторые наиболее часто применяемые директивы и их описание.

**Таблица 20.3. Директивы раздела [*DDInstall.Wdf*] файла INF для драйверов UMDF**

Директива	Обязательная или нет	Описание
<i>UmdfService</i>	Да	Присваивает имя драйверу и указывает на раздел [ <i>Umdf-install</i> ]. Файл INF может содержать несколько директив <i>UmdfService</i> , по одной для каждого сервиса UMDF
<i>UmdfServiceOrder</i>	Да	Указывает порядок, в котором драйверы UMDF должны быть установлены в стеке устройств. Эта директива является обязательной, даже если стек содержит только один драйвер
<i>UmdfDispatcher</i>	Требуется для некоторых типов драйверов	Указывает, куда инфраструктура должна посыпать запросы ввода/вывода после того, как они покинут стек устройств. Эта директива является обязательной для драйверов USB и драйверов, использующих получатели ввода/вывода типа <i>FileHandle</i>
<i>UmdfImpersonationLevel</i>	Нет	Указывает максимальный уровень имперсонации драйвера. Если эта директива опущена, то уровень имперсонации устанавливается равным <i>Identification</i>

Тема имперсонации рассматривается в главе 8.

### Примечание

Для гибридных драйверов, содержащих как драйвер UMDF, так и драйвер KMDF, все драйверы режима ядра в стеке необходимо перечислить с помощью стандартного механизма для драйверов режима ядра или как сервисы, или как драйверы фильтра верхнего или нижнего уровня.

Раздел [*Umdf-install*] обычно содержит директивы, перечисленные в табл. 20.4.

**Таблица 20.4. Директивы раздела [*Umdf-install*] файла INF**

Директива	Обязательная или нет	Описание
UmdfLibraryVersion	Да	Указывает требуемую для драйвера версию UMDF. Номера версий UMDF состоят из трех частей, например, 1.0.0 или 1.5.0
ServiceBinary	Да	Указывает, куда поместить библиотеку DLL драйвера. Для драйверов UMDF это должна быть папка %windir%\System32\Drivers\UMDF
DriverCLSID	Да	Указывает идентификатор CLSID объекта обратного вызова драйвера

Версия соинсталлятора в пакете драйвера должна совпадать с версией библиотеки инфраструктуры в INF-файле. Если компания Microsoft выпускает новую дополнительную версию соинсталлятора, поставщики, использующие новый соинсталлятор, должны отредактировать свои INF-файлы, чтобы указывать новый соинсталлятор и инфраструктуру. В альтернативе, поставщики могут продолжать использовать старую дополнительную версию файла INF и инфраструктуры UMDF и дать пользователям установить новую дополнительную версию инфраструктуры при установке нового драйвера, в котором используется новая дополнительная версия или когда служба обновлений Windows выпустит новую дополнительную версию, чтобы исправить критическую ошибку безопасности.

### Примечание

Все соинсталляторы и содержащиеся в них ресурсы являются подписанными компонентами. Соответствующие сертификаты устанавливаются вместе с Windows или распространяются в пакетах обновлений. Если на целевой системе отсутствует сертификат, которым был подписан соинсталлятор, то установка драйвера завершится неудачей.

## Примеры INF-файлов WDF

С точки зрения пользователя драйверы UMDF и KMDF устанавливаются абсолютно одинаковым способом, и пользователь обычно даже не подозревает, что в действительности это не так. Существенные различия в установке этих двух типов драйверов скрываются от пользователя в большой степени INF-файлом драйверного пакета, который управляет установкой драйвера.

В этом разделе рассматриваются примеры типичных разделов для соинсталлятора UMDF и KMDF файлов INF для драйверов Fx2\_Driver и Osrusbfx2.

## Пример UMDF: INF-файл драйвера Fx2\_Driver

В листинге 20.4 приводится пример разделов соинсталляторов из файла INF для драйвера Fx2\_Driver для сборки драйвера для процессорной архитектуры x86. Краткое описание этих разделов и содержащихся в них директив приводится после листинга.

### Внимание!

Файл INF для образца драйвера Fx2\_Driver в версии набора разработчика WDK, выпущенного с Windows Vista, нельзя применять для установки драйвера на Windows XP. Но пример этого файла, приведенный в листинге 20.4, содержит правильные директивы. Информацию об обновлениях для этого образца драйвера см. в разделе **Developing Drivers with the Windows Driver Foundation** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=80911>.

#### Листинг 20.4. Разделы соинсталляторов INF-файла образца драйвера Fx2\_Driver

```
[OsrUsb_Install.NT]
CopyFiles=UMDriverCopy
Include=WINUSB.INF; Import sections from WINUSB.INF
Needs=WINUSB.NT; Run the CopyFiles & AddReg directives for WinUsb.INF

[OsrUsb_Install.NT.hw]
AddReg=OsrUsb_Device_AddReg
[OsrUsb_Install.NT.Services]
AddService=WUDFRd,0x000001fa,WUDFRD_ServiceInstall
AddService=WinUsb,0x000001f8,WinUsb_ServiceInstall
[OsrUsb_Install.NT.Wdf]
KmddfService=WINUSB, WinUsb_Install
UmdfDispatcher=WinUsb
UmdfService=WUDFOsrUsbFx2, WUDFOsrUsbFx2_Install
UmdfServiceOrder=WUDFOsrUsbFx2
[OsrUsb_Install.NT.CoInstallers]
AddReg=CoInstallers_AddReg
CopyFiles=CoInstallers_CopyFiles
[WinUsb_Install]
KmddfLibraryVersion = 1.5
[WUDFOsrUsbFx2_Install]
UmdfLibraryVersion=1.5.0
DriverCLSID = "{0865b2b0-6b73-428f-a3ea-2172832d6bfc}"
ServiceBinary = "%12%\UMDF\WUDFOsrUsbFx2.dll"
[OsrUsb_Device_AddReg]
HKR,,"LowerFilters",0x00010008,"WinUsb" ;
          FLC_ADDREC_TYPE_MULTI_SZ | FLC_ADDREC_APPEND
[WUDFRD_ServiceInstall]
DisplayName = %Wudfrd DisplayName%
ServiceType = 1
StartType = 3
ErrorControl = 1
ServiceBinary = %12%\WUDFRd.sys
LoadOrderCroup = Base
[WinUsb_ServiceInstall]
DisplayName = %WinUsb_SvcDesc%
ServiceType = 1
```

```

StartType      = 3
ErrorControl   = 1
ServiceBinary  = %12%\WinUSB.sys
[CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,"WudfUpdate_01005.dll",
                           "WdfCoInstaller01005.dll,
WdfCoInstaller","WinUsbCoinstaller.dn"
[CoInstallers_CopyFiles]
WudfUpdate_01005.dll
WdfCoInstaller01005.dll
WinUsbCoinstaller.dll
[DestinationDirs]
UMDriverCopy=12,UMDF; copy to driversMdf
Coinstallers_CopyFiles=11
[UMDriverCopy]
WUDFOsrUsbFx2.dll

```

Большинство разделов в предыдущем примере INF-файла используется драйверами UMDF, но несколько разделов применяются при установке сервиса режима ядра WinUSB, который требуется для USB-драйверов UMDF. Для драйверов других типов эти разделы не нужны. Далее следует описание ключевых разделов и содержащихся в них директив.

◆ Раздел [OsrUsb\_Install.NT].

Директивы `Include` и `Needs` требуются для установки компонента на системы под управлением Windows Vista. Windows XP игнорирует эти директивы.

◆ Раздел [OsrUsb\_Device].

Директива `AddReg` добавляет информацию в реестр, согласно директивам в разделе `[OsrUsb_Device_AddReg]`. В этом разделе WinUSB устанавливается в стеке устройств в качестве драйвера фильтра нижнего уровня.

◆ Раздел [OsrUsb\_Services].

В этом разделе указываются драйверы фильтра режима ядра, требуемые данным устройством.

- Значение `WUDFRd` указывает отражатель UMDF и требуется для всех драйверов UMDF.

Как обсуждалось ранее в этой главе, `WUDFRd` может быть установлен как сервис, верхний драйвер фильтра или нижний драйвер фильтра. Так как драйвер `Fx2_Driver` является функциональным драйвером, то `WUDFRd` устанавливается как сервис.

- Значение `WinUsb` требуется для всех драйверов USB UMDF.

Установка этих компонентов основывается на данных в разделах `WUDFRD_ServiceInstall` и `WinUsb_ServiceInstall` соответственно.

◆ Раздел [OsrUsb\_Wdf].

Это раздел `[DDInstall.WDF]`, упомянутый ранее. Его директивы выполняют следующие задачи.

- Директива `KmdfService` устанавливает сервис WinUSB, который является сервисом режима ядра, используемый USB-драйверами UMDF. Для других типов драйверов эта директива не является обязательной.
- Необязательная директива `UmdfDispatcher` указывает, куда инфраструктура должна посыпать запросы ввода/вывода после того, как они покинут стек устройств. Так как

драйвер Fx2\_Driver является драйвером USB, эта директива должна быть включена в его INF-файл, чтобы обеспечить отправку этих запросов ввода/вывода компоненту WinUSB.

- Директива `UmdfService` задает `WUDFOsrUsbFx2` в качестве имени сервиса и указывает на раздел `[WUDFOsrUsbFx2_Install]` для остальных директив.
- Директива `UmdfServiceOrder` указывает, что драйвер `WUDFOsrUsbFx2` является единственным устанавливаемым драйвером. Если бы этот образец содержал драйверы фильтров, то они бы были перечислены здесь в порядке загрузки, начиная с самого нижнего драйвера фильтра.

◆ Раздел `[OsrUsb_Install.NT.Coinstallers]`.

Это раздел `[DDinstall.Coinstallers]`, упомянутый ранее. Его директивы выполняют следующие задачи:

- директива `AddReg` добавляет требуемую информацию о соинсталляторе в реестр, согласно директивам в разделе `[Coinstaller_AddReg]`;
- директива `CopyFiles` копирует файлы трех соинсталляторов, указанные в разделе `[Coinstaller_CopyFiles]`, в папку, указанную в разделе `[DestinationDirs]`.

◆ Раздел `[WinUsb_Install]`.

В этом разделе указывается номер 1.5 основной версии библиотеки KMDF. Эта библиотека используется сервисом WinUSB.

◆ Раздел `[WUDFOsrUsbFx2_Install]`.

Этот раздел является продолжением директивы `UmdfService`, размещенной ранее в INF-файле. Его директивы выполняют следующие задачи:

- директива `UmdfLibraryVersion` указывает версию 1.5.0 библиотек UMDF;
- директива `DriverCLSID` задает идентификатор CLSID объекта обратного вызова драйвера для образца драйвера `Fx2_Driver` равным `{0865b2b0-6b73-428f-a3ea-2172832d6bfc}`;
- директива `ServiceBinary` указывает, что двоичные файлы драйвера нужно поместить в папку `%Windir%\System32\Drivers\UMDF`. Значение `%12%` является стандартным идентификатором пути `%Windir%\System32\Drivers`.

◆ Раздел `[WUDFRD_ServiceInstall]`.

Этот раздел содержит данные для установки отражателя в качестве сервиса. Эти же настройки можно использовать и в других INF-файлах, которые устанавливают WUDFRd как сервис.

◆ Раздел `[WinUsb_ServiceInstall]`.

Этот раздел содержит данные для установки сервиса WinUSB. Эти же настройки можно использовать и в других INF-файлах, которые устанавливают сервис WinUSB.

◆ Раздел `[DestinationDirs]`.

Этот раздел предоставляет информацию о папках, в которые нужно установить файлы. Значение `12` является стандартным идентификатором папки `%Windir%\System32\Drivers`.

## Пример KMDF: INF-файл драйвера Osrusbfx2

В листинге 20.5 приводится пример разделов соинсталляторов из INF-файла для драйвера Osrusbfx2. Краткое описание этих разделов и содержащихся в них директив приводится после листинга.

### Листинг 20.5. Разделы соинсталляторов INF-файла образца драйвера Osrusbfx2

```
[DestinationDirs] Coinstaller_CopyFiles = 11
[osrusbfx2.Dev.NT.Coinstallers]
AddReg=Coinstaller_AddReg
CopyFiles=Coinstaller_CopyFiles
[Coinstaller_CopyFiles]
wdfcointinstaller01005.dll
[SourceDisksFiles]
wdfcointinstaller01005.dll=1
[Coinstaller_AddReg]
HKR, ,Coinstallers32,0x00010000, "wdfcointinstaller01005.dn,WdfCoinstaller"
[osrusbfx2.Dev.NT.Wdf]
KmdfService = osrusbfx2, osrusbfx2_wdfsect
[osrusbfx2_wdfsect]
KmdfLibraryVersion = 1.5
```

Далее следует описание разделов и содержащихся в них директив.

◆ Раздел [DestinationDirs].

Этот раздел предоставляет информацию о папках, в которые нужно установить файлы. Значение 11 является стандартным идентификатором папки %windir%\System32.

◆ Раздел [OsrUsbFx2.Dev.NT.Coinstallers].

Это раздел [DDInstall.Coinstallers], упомянутый ранее. Он содержит следующие две директивы:

- директива AddReg добавляет требуемую информацию о соинсталляторе в реестр, согласно директивам в разделе [Coinstaller\_AddReg];
- директива CopyFiles копирует файлы, указанные в разделе [Coinstaller\_CopyFiles], в папку, указанную в разделе [DestinationDirs].

◆ Раздел [SourceDisksFiles].

Этот раздел содержит имя DLL соинсталлятора и ее местонахождение. Значение 1 определено в начале INF-файла в разделе [SourceDisksNames] и соответствует установочному диску.

◆ Раздел [Osrusbfx2.Dev.NT.Wdf].

Этот раздел упоминался ранее как [DDInstall.Wdf]. Он назначает сервису режима ядра драйвера имя osrusbfx2.

◆ Раздел [Osrusbfx2\_wdfsect].

Этот раздел упоминался ранее как [wdf-install]. Содержащаяся в нем директива указывает версию 1.5 библиотеки KMDF.

## Подписание и распространение драйверного пакета

Драйверы необходимо подписывать цифровой подписью. Это особенно касается драйверов режима ядра, которые являются доверяемыми компонентами операционной системы и которым разрешается, по существу, неограниченный доступ к системным ресурсам. Цифровая подпись предоставляет пользователям доступ к двум важным аспектам безопасности:

- ◆ верификации происхождения драйверного пакета;
- ◆ гарантии, что пакет не подвергся несанкционированным модификациям.

Драйверы необходимо подписывать цифровой подписью по следующим практическим причинам:

- ◆ Windows Vista и более поздние версии Windows не дадут загрузиться неподписанным драйверам режима ядра на 64-битной системе;
- ◆ подписанные драйверы лучше воспринимаются пользователем;
- ◆ на недавних версиях Windows неподписанные драйверы режима ядра может устанавливать только администратор, и даже ему система выводит диалоговое окно, в котором требует явно подтвердить установку;
- ◆ на Windows Vista для воспроизведения определенных типов оплачиваемого контента (premium content).

В этом разделе дается краткое изложение сущности подписывания драйверов.

### Подписанные файлы каталогов

Непосредственно сами драйверы обычно не подписываются. Вместо этого, драйверный пакет содержит файл каталогов, который играет роль цифровой подписи для всего драйверного пакета. Цифровое подписывание привязывает файл каталогов к конкретному драйверному пакету. Если впоследствии изменится хоть один байт любого компонента пакета, то подпись становится недействительной. Для модифицированного драйверного пакета требуется новый подписанный файл каталогов.

Самую последнюю информацию о требованиях к подписыванию драйверов и способов для этого см. в разделе **Driver Signing Requirements for Windows** (Требования к подписыванию драйверов Windows) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79358>.

Подписанный файл каталогов для драйверного пакета можно получить двумя способами.

- ◆ **Получив логотип Windows.** Драйверы, успешно прошедшие тестирование по программе Windows Logo Program и получившие логотип Windows, также получают файл каталогов для драйверного пакета, подписанный корпорацией Microsoft.

Информацию о процессе тестирования по программе Windows Logo Program см. на Web-сайте этой программы по адресу <http://go.microsoft.com/fwlink/?LinkId=79359>.

- ◆ **Создав свой собственный файл каталогов.** Вы можете получить цифровой сертификат из одного из центров сертификации и с помощью инструментов, предоставляемых в наборе разработчика WDK, создать файл каталогов и подписать его полученным сертификатом.

Более подробную информацию по этому предмету см. в разделе **Creating a Catalog File for a PnP Driver Package** (Создание файла каталога для драйверного пакета PnP) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79360>.

Включение подписанного файла каталогов не является обязательным для тестовых пакетов, предназначенных для 32-битных версий Windows. Тем не менее, тестовые пакеты часто подписываются, чтобы упростить процесс установки или чтобы проверить процедуры установки для подписанных драйверов. Тестовые пакеты можно подписывать тестовым сертификатом, созданным с помощью инструментов, предоставляемых в наборе разработчика WDK.

Дополнительную информацию о создании и установке тестовых сертификатов см. в разделе **Code-Signing Best Practices** (Оптимальные методики по подписыванию кода) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79361>.

## Указание файла каталогов в INF-файле

Подписанный файл каталогов для драйверного пакета указывается директивой **CatalogFile** в разделе **[Version]** файла INF. Так как файл каталогов не используется ни в одном из образцов драйверов USB, то демонстрация включения этого файла дается на примере файла INF для образца драйвера **Featured Toaster**. В нем файлом каталогов для пакета указывается файл **KmdfSamples.cat**.

### Листинг 20.6. Директива CatalogFile в INF-файле для образца драйвера Featured Toaster

```
[Version]
Signature="$WINDOWS NT$"
Class=TOASTER
ClassGuid={B85B7C50-6A01-11d2-B841-00C04FAD5171}
Provider=%MSFT%
DriverVer=02/22/2006,1-0-0.0
CatalogFile=KmdfSamples.cat
```

## Подписывание драйверов BSD

Драйверы BSD устанавливаются во время процесса загрузки операционной системы. Для 64-битных версий Windows Vista, кроме подписанного файла каталогов, цифровая подпись должна быть вставлена в двоичные файлы драйверов BSD.

Это обусловлено тем, что процесс нахождения файла каталогов, чтобы проверить достоверность подписи драйвера, занимает относительно длительное время. Помещение подписей в двоичный файл драйвера ускоряет процесс загрузки. Драйверы BSD должны также иметь и подписанный файл каталогов, который применяется для других целей.

## Распространение драйверного пакета

Собранный драйверный пакет, содержащий подписанный файл каталогов, готов к распространению пользователям. Распространение пакета можно выполнить несколькими способами.

- ◆ Тестовые версии драйверного пакета можно перенести на тестовый компьютер любым удобным способом.

Все образцы драйверов, рассматриваемые в этой главе, были перенесены на тестовый компьютер с помощью USB-диска.

- ◆ Поставляемые версии драйверного пакета обычно записываются на CD или DVD и поставляются вместе с устройством, для которого они предназначены.

Драйверные пакеты можно также выложить для скачивания на Web-сайт. Этот подход особенно полезный при распространении обновлений.

- ◆ Драйверы, успешно прошедшие тестирование по программе Windows Logo Program, можно поместить в содержимое обновлений Windows Update.

При распространении драйверов через программу Windows Update критическим фактором для успешного восприятия продукта конечным пользователем является обеспечить уникальный идентификатор для каждой версии пакета, предназначеннной для определенной аппаратной платформы.

### Внимание!

Корпорация Microsoft настоятельно рекомендует распространять обновления драйверов конечным пользователям посредством Windows Update. Данные, собранные посредством механизма Windows Error Reporting (Служба регистрации ошибок Windows), показывают, что распространение драйверов через Windows Update значительно понижает число фатальных системных сбоев и аппаратных сбоев, вызванных драйверами. Дополнительную информацию на эту тему см. в разделе **Winqual and Distribution Services** (Сервисы Winqual<sup>1</sup> и по дистрибуции) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79362>.

Дополнительную информацию о предоставлении поддержки драйверным продуктам см. в главе 21.

## Установка драйверов

Существует несколько способов для установки и обновления драйверов. Некоторые методы применяются только для установки поставляемых пакетов, другие — только для тестовых установок, а несколько методов можно применить для обоих типов установок.

### Факторы, принимаемые во внимание для тестовых установок

Тестовые установки приходится выполнять многократно на протяжении цикла разработки драйвера, чтобы проверить работу драйвера на разных этапах разработки. Тестовые установочные пакеты, особенно на первоначальной стадии цикла разработки, часто намного больше ограничены, чем поставляемые пакеты. Вызвано это следующими обстоятельствами.

- ◆ Тестовый пакет часто содержит только компоненты, необходимые для установки и работы драйвера.

Дополнительные компоненты, например, сопутствующие пользовательские приложения, в тестовые пакеты часто не включаются.

<sup>1</sup> Windows Quality Online Services. — Пер.

- ◆ Тестовые пакеты обычно или не подписываются, или подписываются тестовым сертификатом.

Но тестовые пакеты для 64-битных версий Windows Vista должны быть подписаны тестовым сертификатом.

- ◆ Тестовые пакеты часто устанавливаются с помощью специальных инструментов, таких как, например, инструмент DevCon из набора разработчика WDK, а не посредством процедур, применяемых при установке пользователем.

Тестовые пакеты для драйверов WDF часто содержат только двоичные файлы драйвера, соинсталляторы WDF и файл INF. Например, для установки тестового пакета образцов драйверов Fx2\_Driver и Osrusbf2:

- ◆ тестовый пакет для драйвера Fx2\_Driver содержит только двоичный файл драйвера, соинсталляторы WDF и файл INF;
- ◆ тестовый пакет для драйвера KMDF содержит двоичный файл драйвера, соинсталлятор KMDF и файл INF.

### Полезная информация

Чтобы ознакомиться с задачами по подписыванию кода, см. раздел **Code-Signing Best Practices** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79361>.

Также см. раздел **Kernel-Mode Code Signing Walkthrough** (Пошаговое рассмотрение процедуры подписывания кода режима ядра) на Web-сайте WHDC по адресу <http://microsoft.com/fwlink/?LinkId=79363>.

## Факторы, принимаемые во внимание для эксплуатационных установок

Эксплуатационные установки выполняются конечными пользователями, чтобы установить в свою систему драйверы для устройства. Драйверные пакеты для эксплуатационных установок должны отвечать следующим требованиям:

- ◆ содержать все поставляемые компоненты;
- ◆ драйверный пакет обычно подписывается рабочим сертификатом, для которого можно установить, что он был выдан доверяемым центром сертификации;

### Примечание

Для 64-битных версий Windows Vista и более поздних версий Windows подписание драйверов является обязательным.

- ◆ пакет устанавливается с помощью инструментов или процедур, доступных пользователю.

Разработчики драйверов также выполняют эксплуатационные установки, обычно чтобы проверить процедуры установки, перед поставкой продукта конечному пользователю. Но для того чтобы не нарушить безопасность рабочего сертификата, эти проверки обычно выполняются, используя тестовый сертификат вместо рабочего.

## Установка драйверов с помощью менеджера PnP

Менеджер PnP автоматически обнаруживает новое устройство Plug and Play и дает пользователю руководящие инструкции в процессе нахождения и установки требуемого драйвера.

Это, скорее всего, наиболее часто применяемый способ установки рабочих драйверов конечными пользователями. Но этот метод плохо подходит для тестовой установки драйверов, т. к. в системе не должно быть установлено более ранних версий драйвера. Если на компьютере уже имеется более ранняя версия драйвера, то менеджер PnP не считает устройство новым и, соответственно, не обновляет, т. е. не устанавливает, драйвер для него.

Далее приводится описание процесса установки любого из функциональных драйверов USB для устройства OSR на "чистый" компьютер. Процесс несколько отличается для разных типов драйвера и для разных версий Windows.

### Установка образца драйвера USB:

1. Если вы еще этого не сделали, выполните сборку любого из образцов драйверов USB. Процедура установки по большому счету одинакова для обоих этих драйверов.
2. Скопируйте драйверный пакет на инсталляционный носитель, например DVD или флэш-диск.
3. Скопируйте драйверный пакет с инсталляционного носителя на тестовый компьютер.
4. Подключите устройство OSR USB Fx2, вставив его кабель в разъем USB.

Менеджер PnP должен обнаружить устройство и вывести на экран последовательность диалоговых окон с инструкциями по установке.

Подробности процесса установки отличаются в зависимости от того, для какой версии Windows устанавливается драйвер. Они также зависят от того, подписан драйвер или нет, а если подписан, то каким сертификатом.

Эта процедура применима только при первом физическом подключении устройства. Если же в системе уже установлен драйвер для устройства, то менеджер PnP не считает устройство новым и не запускает процесс установки драйвера. Если на тестовом компьютере уже установлена более старая версия драйвера, то обновление драйвера выполняется с помощью Device Manager (Диспетчер устройств). Эта процедура рассматривается в разд. *"Обновление драйверов с помощью Диспетчера устройств"* далее в этой главе.

### Полезная информация

Информация о том, каким образом менеджер PnP выполняет установку драйверов, см. в разделе **How Setup Selects Drivers** (Как утилита установки выбирает драйверы) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79364>. Также см. раздел **Device Installation Rules for Windows Vista** (Правила по установке устройств для Windows Vista) на Web-сайте WHDC по адресу <http://microsoft.com/fwlink/?LinkId=79365>.

Узнать больше, о том, как менеджер PnP выполняет подписывание драйвера, можно в уже упоминавшемся разделе **Code-Signing Best Practices** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79361>.

## Установка драйверов с помощью SPInst или DIFxApp

Набор инструментов DIFx в WDK содержит два свободно распространяемых настраиваемых приложения для установки драйверов — DPInst и DIFxAPP. С помощью этих инструментов можно выполнять как тестовую, так и эксплуатационную установку драйверов, а также обновлять уже установленные драйверы. Эти приложения дают пользователям инструкции по процессу установки правильного драйвера. Они также помещают установленный

драйвер в список служебной программы **Programs and Features** (которая на более ранних версиях Windows называется **Add or Remove Programs** (Установка и удаление программ)).

- ◆ DPInst — это настраиваемое приложение для установки драйверных пакетов. DPInst обычно применяется лишь тогда, когда устанавливается только драйверный пакет.
- ◆ DIFxApp — это настраиваемый установщик Windows для специализированных установок. DIFxApp обычно применяется для установки драйверов, которые распространяются совместно с пакетом приложения.

Оба приложения можно настроить для удаления драйверов.

#### **Полезная информация**

Приложения DPInst и DIFxApp являются рекомендованными решениями для установки драйверов, как описано в разделе **Using Driver Install Frameworks (DIFx)** (Применения инфраструктур по установке драйверов) документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79366>.

## **Установка драйверов с помощью специализированного установочного приложения**

Подходы, описанные в предыдущих разделах, хорошо подходят для многих установочных сценариев. Но некоторые устройства имеют специальные требования по установке, которые эти подходы не могут удовлетворить. Также часто бывает, что поставщики предпочитают использовать собственные фирменные процедуры установки. В таком случае, самым лучшим подходом будет реализовать специализированное установочное приложение. Это обычное приложение пользовательского режима, похожее на мастера, которое устанавливает драйвер и выполняет другие задачи, связанные с установкой. Это приложение может также и удалить драйвер.

#### **Полезная информация**

Инструменты DIFx также содержат интерфейс API (DIFxAPI), который рекомендуется использовать для реализации специализированных установочных приложений для драйверов. Дополнительную информацию на эту тему см. в разделе **Writing a Device Installation Application** (Создание приложения для установки устройства) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79367>.

## **Установка и обновление драйверов с помощью инструмента DevCon**

DevCon — это инструмент командной строки, входящий в набор разработчика WDK, для управления устройствами компьютера, включая установку и обновление драйверов. Этот инструмент предназначен для использования только разработчиками.

Обычно DevCon применяется для установки или обновления драйверов на тестовом компьютере. Для установки драйвера применяется следующая команда:

```
Devcon install INF_FileName Hardware_ID
```

Здесь *INF\_FileName* — это путь и имя INF-файла драйверного пакета. Если путь не указан, то DevCon полагает, что INF-файл находится в текущей папке. *Hardware\_ID* — это идентификатор аппаратного обеспечения устройства.

### Примечание

Для этой команды DevCon двоичный файл драйвера, файл INF и соинсталляторы должны находиться в одной папке.

Для обновления драйвера в приведенном примере команды просто замените слово `install` словом `update`.

### Полезная информация

Набор разработчика WDK содержит отдельную версию DevCon для каждой поддерживающей процессорной архитектуры в подпапках папки `%WDK%\tools\devcon`. Дополнительную информацию по этому инструменту см. в его документации в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79777>.

## Обновление драйверов с помощью Диспетчера устройств

При тестировании и отладке драйверы приходится устанавливать по несколько раз, поэтому установка обновленных сборок драйвера на тестовые компьютеры является нормальным событием процесса разработки драйвера. Одним из удобных способов обновления драйверов является Device Manager (Диспетчер устройств). Пользователи также могут обновлять драйверы с помощью Диспетчера устройств, но обычно им пользуются только сравнительно опытные пользователи.

Процедура обновления драйверов с помощью Диспетчера устройств:

1. Запустите Диспетчер устройств.
2. В окне Диспетчера устройств щелкните правой кнопкой по требуемому устройству и в открывшемся контекстном меню выберите пункт **Update Driver Software** (Обновить драйвер).
3. Откроется окно Мастера обновления оборудования. Следуя его инструкциям, укажите местонахождение драйверного пакета и установите новый драйвер.

## Удаление драйверов

Windows устроена таким образом, что пользователям обычно не требуется ничего знать об удалении драйверов. Процесс удаления драйвера обычно выполняется невидимо для пользователя и без его участия в нем. Но разработчикам необходимо понимать процесс удаления драйверов по двум причинам:

- ◆ они должны реализовывать установочные приложения таким образом, чтобы можно было осуществить корректное удаление драйвера;
- ◆ иногда им необходимо удалять драйверы явным образом, чтобы подготовить компьютер для новых сценариев тестирования.

Например, при установленной на тестовом компьютере старой версии драйвера нельзя протестировать установку его новой версии менеджером PnP.

Дополнительную информацию по удалению драйверов см. в разделе **Uninstalling Drivers and Devices on Windows Vista** (Удаление драйверов и устройств в Windows Vista) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80089>.

## Процесс установки драйверов

Прежде чем приступить к разработке процедуры удаления драйвера, необходимо понимать процесс, применяемый Windows для установки драйвера WDF, и каким образом операция удаления может отменить разные части этого процесса. Процесс установки драйверов на Windows Vista осуществляется в четыре этапа. На каждом этапе выполняется одна или несколько операций, которые могут быть отменены позже процедурой удаления. Этапы пронумерованы только с целью облегчения ссылок на них в дальнейшем. Они не обязательно выполняются в порядке нумерации.

### Этап 1. Установка требуемой инфраструктуры WDF (если необходимо)

Если в системе установлена устаревшая версия инфраструктуры, или не установлено никакой версии, то соинсталлятор устанавливает текущую версию UMDF или KMDF, в зависимости от того, какая из этих инфраструктур требуется для драйвера.

### Этап 2. Создание узла devnode для устройства

Менеджер PnP создает узел devnode для устройства и ассоциирует его с соответствующим деревом устройств.

### Этап 3. Разворачивание драйверного пакета в хранилище драйверов

На данном этапе установки выполняются две операции.

- ◆ Все драйверы в пакете копируются в хранилище драйверов (driver store), которое управляетя DIFx. В этом процессе распаковываются все упакованные файлы и создается дубликат структуры каталогов носителя исходных кодов пакета. Внутренняя база данных менеджера PnP автоматически обновляется метаданными драйверного пакета.
- ◆ Файл INF копируется в папку %Windir%\Inf.

### Этап 4. Установка драйвера

На том этапе выполняется несколько операций:

- ◆ двоичные файлы копируются в их целевые папки;
- ◆ вызывается диспетчер управления сервисами;
- ◆ обновляются разделы реестра;
- ◆ компоненты регистрируются в COM (только для драйверов UMDF).

## Действия по удалению драйвера

При удалении драйвера выполняются три основные операции: удаляется устройство, удаляется драйверный пакет и удаляются двоичные файлы драйвера. В этом разделе дается краткое описание этих трех основных операций.

### Примечание

Стандартные методы и инструменты для удаления драйверов, рассматриваемые в этой главе, не могут удалять инфраструктуры. Инфраструктуры нельзя удалять, а только обновлять по мере необходимости. Таким образом, обеспечивается их наличие для других драйверов.

веров WDF в системе. В Windows Vista и более поздних версиях Windows инфраструктуры являются составной частью операционной системы и не могут быть удалены в принципе. Таким образом, исключается случайное их удаление и обеспечивается их наличие для системных компонентов, зависящих от них.

## Удаление устройства

В этой операции удаляется узел devnode, ассоциированный с устройством. После завершения операции удаления устройства экземпляр устройства больше не существует, но драйверный пакет остается в хранилище драйверов. Если устройство извлечь из разъема, а потом снова вставить, то менеджер PnP будет рассматривать его, как новое устройство.

Эта операция удаления выполняется автоматически менеджером PnP и отменяет этап 2 процесса установки и некоторые действия этапа 4.

Когда менеджер PnP удаляет устройство, то он просто удаляет подмножество состояний системы, которые были созданы во время установки. Драйверный пакет и двоичные файлы остаются там же, где они и были, и разделы реестра, созданные установщиком класса и со-инсталлятором, и другие действия, выполненные над реестром, не отменяются. Но этой операции удаления достаточно, чтобы предотвратить Windows от загрузки драйвера для всех распространенных пользовательских сценариев.

## Удаление драйверного пакета из хранилища драйверов

В этой операции файлы пакета удаляются из хранилища драйверов, а также удаляются соответствующие метаданные из внутренней базы данных менеджера PnP. Также удаляется INF-файл пакета из папки %Windir%\Inf. После удаления пакета из хранилища драйверов он больше недоступен для установки. Чтобы установить драйвер сейчас, необходимо опять выполнить развертывание пакета в хранилище драйверов с первоначального источника, например, оптического носителя, сетевого каталога или Windows Update.

Эта операция удаления отменяет действия этапа 3 процесса установки. Менеджер PnP выполняет эти операции автоматически, если пользователь выбрал опцию **Delete the driver software for this device** (Удалить драйвер для этого устройства) в начале процесса удаления драйвера.

### Внимание!

Не удаляйте драйверный пакет из хранилища драйверов вручную. Это может вызвать несоответствие между файлом INF, каталогом хранилища драйверов и драйвером в хранилище драйверов, в результате чего может оказаться невозможным выполнить повторное развертывание драйверного пакета в хранилище драйверов.

## Удаление двоичных файлов драйвера

В этой операции удаляются двоичные файлы драйвера из установочной папки, обычно %Windir%\System32\Drivers.

Эта операция удаления отменяет некоторые действия этапа 4 процесса установки. Менеджер PnP не поддерживает эту операцию, и ее необходимо выполнять с использованием инструментов DIFx. Инструменты DIFx проверяют на соответствие между файлом в каталоге установки и файлом в хранилище драйверов; они не используют в своей работе путь и имя файла. Поэтому корпорация Microsoft настоятельно рекомендует, чтобы этот способ удаления

драйверов был основан только на применении инструментов DIFx, которые предназначены для обеспечения надежных процедур удаления драйверов.

### Внимание!

Инструменты DIFx не отслеживают число устройств, зависящих от двоичного файла драйвера, и использование этого файла другими компонентами. При удалении устройства соответствующие двоичные файлы драйверов могут находиться в использовании другими устройствами или приложениями, поэтому их удаление обычно вызывает какой-либо сбой. Поэтому, прежде чем удалять двоичные файлы драйвера, деинсталлятор должен удостовериться в том, что они не используются никакими другими компонентами системы. Лишь в этом случае эти файлы можно безопасно удалить.

## Инструменты для удаления устройств и драйверов

Существует два основных способа для удаления устройств и драйверов: с применением Диспетчера устройств Windows и с применением инструментов DIFx. В этом разделе рассматривается использование этих средств для реализации операций удаления, рассмотренных в предыдущем разделе. Предоставляется краткое описание этих инструментов и их возможности.

### Диспетчер устройств

С помощью Диспетчера устройств можно деинсталлировать драйвер, но не удалить его двоичные файлы. В Windows Vista Диспетчер устройств находится в Панели управления. В более ранних версиях Windows доступ к Диспетчеру устройств можно получить посредством приложения Administrative Tools (Администрирование) в Панели управления.

### Примечание

Пользователь, желающий удалить устройство или деинсталлировать драйвер с помощью Диспетчера устройств, должен быть членом локальной группы **Администраторы** или эквивалентной группы.

## Инструменты DIFx

Инструменты DIFx удаляют как драйверный пакет, так и двоичные файлы драйвера. После удаления драйвера с помощью инструментов DIFx устройство обычно удаляется с помощью Диспетчера устройств.

### Внимание!

Для удаления драйвера необходимо использовать тот же самый инструмент, с помощью которого он был установлен. Например, драйвер, установленный с помощью инструмента DIFxApp, нельзя удалять с помощью инструмента DPInst.

Для создания деинсталляторов драйверов корпорация Microsoft рекомендует применять только инструменты DIFx. Приложения, использующие инструменты DIFx для удаления драйверов, будут работать корректно с будущими версиями Windows. Хотя другие подходы могут работать успешно с текущими версиями Windows, нет гарантии, что они будут работать корректно с будущими версиями.

## Инструмент DevCon

Инструмент DevCon обычно применяется для удаления устройств на тестовых компьютерах под управлением более ранних версий Windows, чем Windows Vista.

**Чтобы удалить драйвер с помощью инструмента DevCon, выполните следующую команду:**

```
devcon remove ID
```

Здесь *ID* — это идентификатор аппаратного обеспечения устройства. Эта команда удаляет стек устройств и узел devnode, но не удаляет двоичные файлы драйвера или драйвер и его файл INF из хранилища драйверов.

**Чтобы удалить драйвер и файл INF из хранилища драйверов, выполните следующую команду:**

```
devcon dp_delete INF_File
```

Здесь *INF\_File* указывает путь и имя INF-файла пакета, который расположен в папке %Windir%\Inf.

## Поиск и удаление проблем с установкой драйверов WDF

Далее приводится несколько советов по поиску и удалению проблем, возникающих с установкой драйверов, а также наиболее распространенные ошибки и возможные способы их устранения. Прежде чем приступить к исследованию проблемы с установкой устройства, необходимо иметь следующие компоненты.

- ◆ Драйверный пакет, включая файл INF и все файлы в хранилище драйверов.  
Например, драйверный пакет образца драйвера Fx2\_Driver находится в папке %windir%\System32\DriverStore\FileRepository\osrusbfx2.inf\_4e4bba4c. Цифры, следующие после имени INF-файла, генерируются автоматически, чтобы создать сильное имя, что исключает возможность конфликта имен.
- ◆ Журналы регистрации ошибок.  
Дополнительную информацию на эту тему см. в разд. "Журналы регистрации ошибок установки драйверов" далее в этой главе.
- ◆ Информацию обо всех сообщениях об ошибках и месте их возникновения.
- ◆ Все коды ошибок устройства, предоставленные Диспетчером устройств.  
Коды ошибок для Windows Vista такие же, как и для Windows XP. Список кодов ошибок Диспетчера устройств и предлагаемых решений см. в статье 310123, "Explanation of error codes generated by Device Manager" ("Объяснения кодов ошибок, генерируемых Диспетчером устройств"), в Microsoft Knowledge Base (база знаний Microsoft) по адресу <http://support.microsoft.com/kb/310123>.
- ◆ Идентификатор аппаратного обеспечения (Hardware ID) устройства.  
Чтобы получить эту информацию, запустите Диспетчер устройств и откройте диалоговое окно **Properties** (Свойства) драйвера. Идентификатор аппаратного обеспечения находится на вкладке **Details** (Сведения) этого окна.

## Поиск и исправление ошибок установки с помощью отладчика WinDbg

Установка устройств иногда завершается неудачей по причине проблем с кодом загрузки и запуска драйвера. Для драйверов UMDF эти ошибки обычно случаются в методе

`IDriverEntry::OnDeviceAdd`. А для драйверов KMDF такие ошибки обычно происходят в функции `DriverEntry` или в процедуре `EvtDriverDeviceAdd`.

Для выявления и устранения таких ошибок установите точки прерываний в проблемных процедурах, после чего используйте WinDbg, чтобы определить причину ошибок.

Дополнительную информацию об отладке драйверов и отладчике WinDbg см. в главе 22.

## Журналы регистрации ошибок установки драйверов

Все компоненты, принимающие участие в установке драйвера, протоколируют информацию об установке в файл журнала. Эти файлы можно просмотреть на наличие информации, относящейся к WDF и устанавливаемому драйверу.

Все версии Windows, поддерживающие WDF, имеют следующие файлы журнала протоколирования установки.

- ◆ **Файл журнала SetupAPI.** Содержит информацию, которую SetupAPI записывает при каждой установке драйвера в систему.

Этот журнал полезный для определения, была ли предпринята попытка установки устройства. Если в журнале имеются данные, указывающие на вызов соинсталляторов, проверьте журнал протоколирования операции установки для дополнительной информации. Если в журнале имеются признаки выполнения системой обновления драйвера, проверьте журналы протоколирования обновления WDF.

Windows Vista имеет два журнала SetupAPI: журнал установки устройства `%windir%\inf\Setupapi.dev.log` и журнал установки приложения `%windir%\inf\Setupapi.app.log`.

Более ранние версии Windows имеют один журнал SetupAPI — `%systemroot%\setupapi.log`

- ◆ **Журнал протоколирования операции установки.** Содержит отладочные сообщения от соинсталляторов WDF.

Находится в файле `%windir%\setupact.log`.

- ◆ **Журналы протоколирования обновления WDF.** Содержат информацию об ошибках при установке WDF (имеются только в Windows Vista).

В журнале протоколирования обновлений KMDF регистрируется информация о событиях и ошибках, происходящих при установке драйверов KMDF. Этот журнал находится в файле `%windir%\wdfMMmmminst.log`, где *MM* указывает основную версию инфраструктуры, а *mm* — дополнительную.

В журнале протоколирования обновлений UMDF регистрируются сообщения соинсталлятора UMDF. Находится в файле `%windir%\temp\wudf_update.log`.

- ◆ **Журнал протоколирования ошибок установки.** Содержит сообщения об ошибках при установке.

Находится в файле `%windir%\setuperr.log`.

- ◆ **Журнал системных событий.** Содержит информацию об ошибках, происходящих при динамической привязке драйвера KMDF к библиотеке.

Этот журнал можно просмотреть, открыв консоль **Управление компьютером** и выбрав элемент **Просмотр событий**.

Объем информации, записываемой для каждого события в журнал SetupAPI, можно контролировать с помощью параметра реестра. Иногда бывает полезным повысить уровень прото-

колирования для всех приложений установки. Ошибки, возникающие при установке драйвера, в действительности могут порождаться ошибкой установки какого-либо другого драйвера в стеке устройств. Повышение уровня протоколирования может помочь в обнаружении таких ошибок.

Дополнительную информацию о протоколировании SetupAPI см. в разделе **Troubleshooting Device Installation** (Поиск и устранение ошибок при установке устройства) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79370>.

### Полезная информация

В журналах SetupAPI и протоколирования операции установки ведется кумулятивная регистрация всех установок, и их объем может быть довольно большим. В случае воспроизведенной ошибки, переименуйте файлы журналов и переустановите драйвер. Система создаст новые журналы для протоколирования SetupAPI и операции установки, в которых будут только данные для вашего драйвера.

## Распространенные ошибки при установке драйверов WDF

При установке драйверов могут возникнуть разного рода ошибки. Большая часть этих ошибок — общего типа, например, несоответствие между файлами, указанными в INF-файле, и файлами, в действительности содержащимися в пакете. В этом разделе рассматриваются некоторые наиболее распространенные ошибки при установке драйверов WDF и возможные решения для них.

Общую информацию об ошибках при установке драйверов см. в разделе **Guidelines for Using SetupAPI** (Руководство по использованию SetupAPI) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79371>.

### Фатальные ошибки при установке

Эти ошибки могут возникнуть вследствие нескольких причин. Наиболее распространеными причинами, среди прочих, являются следующие.

- ◆ **Неправильная версия соинсталлятора.** Свободные версии соинсталлятора можно использовать только со свободными версиями Windows. То же самое относится и к проверочным версиям. Неправильный соинсталлятор нужно удалить вручную из папки %windir%\system32\.

При попытке использования неправильного соинсталлятора, журнал Setupact.log будет содержать сообщение наподобие следующего:

```
Message A
WdfCoInstaller: [12/08/2006 22:28.25.834]
Update process returned error code:error(1603)
Fatal error during installation.

Message B
Possible causes are running free version of coinstaller on checked
version of OS and vice versa.

Message C
WdfCoInstaller: [12/08/2006 22:28.25.864] Final status: error(1603)
Fatal error during installation.
```

- ◆ Службы криптографии не запущены. Откройте Панель управления, откройте окно Администрирование, а в нем — консоль Службы. Найдите элемент Службы криптографии и запустите службу.
- ◆ Система не доверяет сертификату, которым подписано обновление. Эта проблема обычно возникает, когда на тестовой системе не установлен тестовый корневой сертификат.
- ◆ В файле INF повторяются директивы. Из-за ошибки в версии 1.1 KMDF установка завершается неудачей, если файл INF содержит больше чем одну директиву KmdfService или KmdfLibraryVersion.

Эта ситуация может возникнуть с драйверами фильтра, если сервисы драйвера фильтра и функционального драйвера включены в один и тот же INF-файл. Версия 1.5 KMDF этой ошибки не имеет, и следует пользоваться нею.

## Коды ошибок менеджера PnP

Если при установке устройства менеджером PnP возникает ошибка, он возвращает код ошибки, чтобы указать причину проблемы. Устройство будет присутствовать в списке Диспетчера устройств, но будет обозначено желтым восклицательным знаком, обозначающим неудачную установку. Для просмотра кодов ошибок откройте диалоговое окно Свойства в Диспетчере устройств. В этом разделе рассматриваются коды двух часто встречающихся ошибок для драйверов WDF.

С полным списком кодов ошибок для менеджера PnP можно ознакомиться в разделе **Device Manager Error Messages** (Коды ошибок Диспетчера устройств) документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80068>.

**Код ошибки 37.** Этот код указывает на проблему или с соинсталлятором, или в функции DriverEntry драйвера. Если ошибка возникла по одной из этих причин, журнал протоколирования операции установки может содержать одно из следующих сообщений:

```
Final status: error(0) The operation completed successfully.
GetLatestInstalledVersion install version major 0x1,
minor 0x1 is less than or equal to latest major 0x1,
minor 0x1, asking for post processing
WdfCoinstaller: DIF_INSTALLDEVICE: Post-Processing
```

Наиболее вероятная причина этой ошибки — проблема в функции DriverEntry драйвера. Установите контрольную точку в этой функции и используйте отладчик WinDbg, чтобы найти проблему.

**Код ошибки 31.** Эта ошибка всегда вызывается какой-либо проблемой в драйвере. Часто эта ошибка происходит, когда функция EvtDriverDeviceAdd или метод IDriverEntry::OnDeviceAdd возвращает статус иной, нежели STATUS\_SUCCESS.

**Другие ошибки при установке.** Причины других ошибок при установке должны быть очевидными по записям в журнале протоколирования установки. Далее приводится несколько типичных примеров.

- ◆ Искаженный раздел [wdf] файла INF.

Версия заголовочного файла, использованного при компиляции драйвера, не совпадает с версией библиотеки.

◆ Плохое декодирование имен разделов в файле INF.

Если применяется спецификатор, то он должен применяться во всех релевантных разделах. Например, будет неправильным присвоить разделам сервисов и соинсталляторов в файле INF имена DDinstall.NT.Services и DDinstall.Coinstallers соответственно. Эту ошибку можно исправить, переименовав раздел соинсталляторов в DDinstall.NT.Coinstallers. Такие ошибки можно уловить с помощью инструмента CheckInf.

◆ Несоответствие версии соинсталлятора версии указанной в директиве KrmdfLibraryVersion или UmdfLibraryVersion.

Эти версии должны быть одинаковыми.

◆ Диспетчер устройств пользуется устарелым файлом INF вместо обновленного.

Этой проблемы можно избежать двумя способами. Удалите старые версии файлов oem\*.inf и oem\*.pnf из папки %windir%\inf, чтобы старый файл INF больше не был доступным. Или же укажите путь поиска к новому драйверу, вместо того, чтобы позволить Диспетчеру устройств применить путь поиска по умолчанию.

# ГЛАВА 21

## Инструменты для тестирования драйверов WDF

Всесторонне тестирование на всех стадиях разработки является абсолютно необходимым для создания надежного, высококачественного драйвера. Прежде чем отправлять драйвер пользователям, очень важно найти и исправить как можно больше ошибок. Особенно это касается драйверов режима ядра, ошибки в которых имеют намного большую вероятность вызвать фатальный сбой или зависание системы.

В этой главе дается краткий обзор инструментов для тестирования и верификации драйверов WDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b>	
Инструменты WDK для тестирования и верификации драйверов	%wdk%\tools
Инструмент Application Verifier	<a href="http://go.microsoft.com/fwlink/?LinkId=79601">http://go.microsoft.com/fwlink/?LinkId=79601</a>
Инструмент Driver Verifier	%windir%\System\Verifier.exe
Инструменты KernRate и KRVView	<a href="http://go.microsoft.com/fwlink/?LinkId=79779">http://go.microsoft.com/fwlink/?LinkId=79779</a>
<b>Документация WDK</b>	
Раздел "Driver Development Tools" <sup>1</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79298">http://go.microsoft.com/fwlink/?LinkId=79298</a>
Раздел "Debugging Framework-based Driver" <sup>2</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79790">http://go.microsoft.com/fwlink/?LinkId=79790</a>
Раздел "Handling Driver Failures" <sup>3</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79794">http://go.microsoft.com/fwlink/?LinkId=79794</a>
Раздел "Windows Device Testing Framework" <sup>4</sup>	<a href="http://go.microsoft.com/fwlink/?LinkId=79785">http://go.microsoft.com/fwlink/?LinkId=79785</a>
<b>Прочее</b>	
Раздел Windows Error Reporting: Getting Started <sup>5</sup> на Web-сайте WHDC	<a href="http://go.microsoft.com/fwlink/?LinkId=79792">http://go.microsoft.com/fwlink/?LinkId=79792</a>

<sup>1</sup> Инструменты для разработки драйверов. — *Пер.*

<sup>2</sup> Отладка драйверов WDF. — *Пер.*

<sup>3</sup> Обработка сбоев драйверов. — *Пер.*

<sup>4</sup> Среда тестирования устройств Windows. — *Пер.*

<sup>5</sup> Начало работы со службой регистрации ошибок Windows. — *Пер.*

## Начало работы по тестированию драйверов

Если вы никогда раньше не занимались тестированием драйверов, то в добавление к системе для разработки вам будет необходимо подготовить тестовую систему. В этом разделе рассматриваются основные системные требования к тестовой системе, а также проводится обзор инструментов для тестирования драйверов, предоставляемых корпорацией Microsoft.

Подробную информацию об отладке драйверов KMDF и UMDF см. в главе 22.

### Выбор тестовой системы

Драйверы KMDF необходимо тестировать на другой системе, чем система, применяемая для их разработки. Это позволяет избежать воздействия отладчика на нормальную работу операционной системы, что могло бы повлиять на работу тестируемого драйвера. Это также позволяет сэкономить время на перезагрузках и восстановлениях после системных сбоев, которые являются неизбежной частью процесса разработки.

Тестирование и отладку драйверов UMDF можно выполнять на одной системе; для отладчика не требуется отдельная система.

Тестовая система должна иметь следующие возможности.

- ◆ Последовательный порт или порт IEEE 1394 для отладки в режиме ядра.
- ◆ За исключением тестирования драйверов, предназначенных исключительно для однопроцессорных систем, тестовая система должна быть многопроцессорной, или если однопроцессорной, то, по крайней мере, поддерживать гиперпотоковость (hyperthreading).

Отладку проблем, специфичных для многопроцессорных систем, нельзя выполнять на однопроцессорной системе. Теперь, когда даже компьютеры низшего класса поддерживают гиперпотоковость, очень важно протестировать разрабатываемый драйвер для многопроцессорных сценариев. Отличной тестовой платформой будет многопроцессорная система с процессорной архитектурой x64, т. к. такая платформа поддерживает как 32-битные, так и 64-битные версии Windows.

Непременно протестируйте разрабатываемый драйвер для однопроцессорных сценариев. На многопроцессорных системах это легче всего сделать, выбрав опцию /ONECPU в файле Boot.ini или опцию onecpu BCDEdit в настройках загрузки операционной системы.

- ◆ Сменные жесткие диски, что облегчает тестирования для разных версий Windows.
- ◆ Если устройство поддерживает DMA, то тестовая машина должна иметь больше чем 4 Гбайт системной памяти.

Такой объем памяти может казаться излишним, но это может пригодиться при определении причин проблем, возникающих при адресации расширенной памяти. Такие проблемы трудно определить, если система не оборудована достаточным объемом памяти.

На тестовой системе должна быть установлена проверочная версия Windows, под которую разработан драйвер. Это один из самых мощных инструментов для тестирования и отладки драйверов. Но установка полной проверочной версии Windows может вызвать значительный отрицательный эффект на производительность системы. Поэтому, вместо установки полной проверочной сборки, можно выполнить сборку драйвера в проверочной среде сборки и установить только проверочную версию стека драйвера. Также можно только заменить файлы ntoskrnl.exe и hal.dll их проверочными версиями.

Дополнительную информацию по теме см. в разделе **Installing Just the Checked Operating System and HAL** (Установка проверочных версий только операционной системы и HAL) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79774>.

Использование проверочных и свободных сборок Windows для разработки драйверов WDF рассматривается в *главе 1*.

Если драйвер предназначается для применения с несколькими версиями Windows, но у вас ограниченное количество тестовых систем, можно установить по несколько версий Windows на каждую тестовую машину. Это позволит вам сэкономить время, избавив от необходимости устанавливать и переустанавливать операционную систему для каждого этапа тестирования.

## Обзор инструментов для тестирования драйверов WDF

Корпорация Microsoft предоставляет несколько инструментов для тестирования драйверов во время их разработки. Большинство из этих инструментов включены в набор разработчика драйверов WDK. Эти инструменты можно разбить на следующие две основные категории.

- ◆ Инструменты для статического анализа кода, которые анализируют код на ошибки, не исполняя код.

К инструментам статического анализа относятся инструмент PREfast for Drivers, с помощью которого можно проверять как драйверы KMDF, так и драйверы UMDF, и инструмент Static Driver Verifier (SDV), который используется для тестирования только драйверов KMDF.

- ◆ Инструменты для динамического анализа кода, которые исполняют установленный драйвер по всему диапазону его возможностей, с целью активировать возможные ошибки и вызвать сбой драйвера.

К этой категории инструментов тестирования относятся инструменты Driver Verifier, KMDF Verifier, UMDF Verifier и Application Verifier. Эти четыре инструмента рассматриваются подробно далее в этой главе.

## Методы трассировки для тестирования драйверов

При трассировке действия драйвера записываются в формате, позволяющем анализировать их впоследствии с помощью отладчика или другого инструмента. Сообщения трассировки можно также выводить для просмотра сразу же при их выдаче. Обязательно воспользуйтесь следующими методами при тестировании драйверов.

- ◆ Препроцессор трассировки Windows.

Препроцессор WPP упрощает использование и расширяет возможности утилиты Event Tracing for Windows. Эта утилита режима ядра для протоколирования трассировки регистрирует как события режима ядра, так и события, определенные в приложении. Генерирование сообщений трассировки предоставляет относительно легкий способ получения и сохранения информации о работе драйвера. Утилита позволяет определять флаги трассировки, соответствующие типам информации, которую вы считаете наиболее полезной при отладке.

Тема трассировки WPP подробно рассматривается в *главе 11*.

#### ◆ Журнал KMDF.

Инфраструктура KMDF содержит внутренний регистратор трассировки, основанный на трассировке WPP. Регистратор отслеживает прохождение пакетов IRP по инфраструктуре и соответствующих объектов `WDFREQUEST` по драйверу. Он также отслеживает изменения в состояниях Plug and Play и энергопотребления и другие внутренние изменения в инфраструктуре. Регистратор ведет запись недавних событий трассировки для каждого экземпляра драйвера — для каждого драйвера KMDF ведется собственный журнал. Журнал KMDF можно просматривать и сохранять во время интерактивной отладки с помощью расширений отладчика WDF.

Подробно регистратор KMDF рассматривается в [главе 22](#).

## Инструменты PREfast и SDV

Инструмент статического анализа исходного кода PREfast for Drivers позволяет отловить определенные типы ошибок, обнаружение которых представляет трудность для компилятора. Инструмент PREfast можно применять сразу же после компиляции исходного кода. Для этого не требуется ни выполнять компоновку, ни исполнять код.

В работе инструмент PREfast эмулирует исполнение возможных ветвей кода одна функция за другой, включая ветви кода, которые редко исполняются при реальной работе драйвера. Инструмент PREfast проверяет каждую возможную ветвь кода на соответствие правилам, которые определяют возможные ошибки или некачественно написанный код, и протоколирует предупреждения, вызванные кодом, возможно нарушающим эти правила. Инструмент PREfast можно применять для тестирования как драйверов режима ядра, так и других компонентов режима ядра.

Подробная информация об использовании инструмента PREfast для верификации исходного кода драйверов WDF излагается в [главе 23](#).

Инструмент для статической верификации SDV применяется в конце цикла разработки для проверки исполняемых драйверов, работа над которыми подходит к фазе тестирования. Инструмент SDV выполняет более глубокое тестирование, чем PREfast, и часто обнаруживает дополнительные ошибки, после того, как были исправлены ошибки, обнаруженные инструментом PREfast. Начиная с набора разработчика драйверов WDK для Windows Server Longhorn, инструмент SDV можно применять для тестирования драйверов KMDF и драйверов режима ядра WDM.

Инструмент SDV исследует ветви кода драйвера, символически исполняя исходный код, подвергая тестированию код в ветвях, которые не замечаются при обычном тестировании. При верификации инструмент SDV исследует все действительные ветви кода драйвера и применяемого драйвером кода библиотек и пытается доказать, что драйвер нарушает правила SDV. Если ему не удается доказать нарушение, он сообщает, что драйвер удовлетворяет требованиям правил и успешно прошел верификацию. Правила SDV включают правила общего назначения, применимые к любому драйверу режима ядра, и правила, специфичные для KMDF, применимые только к драйверам KMDF.

Подробности тестирования драйверов WDF с помощью инструмента SDV излагаются в [главе 24](#).

## Другие инструменты для тестирования драйверов

Кроме PREfast, SDV и других инструментов для верификации драйверов, описанных в этой главе, набор разработчика драйверов WDK содержит многие другие инструменты. В этом

разделе описывается несколько часто применяемых инструментов для тестирования драйверов, предоставляемых в этом наборе. Если не указывается иное, эти инструменты находятся в папке %wdk%\tools.

Дополнительную информацию по этому вопросу см. в разделе **Driver Development Tools** (Инструменты для разработки драйверов) на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79298>.

## Инструмент INF File Syntax Checker (ChkINF)

Инструмент INF File Syntax Checker, или ChkINF, является одним из лучших способов для получения работоспособного установочного пакета драйвера. Инструмент ChkINF представляет собой сценарий на языке Perl, поэтому для работы с ним требуется установить один из интерпретаторов Perl, указанных в документации набора разработчика драйверов WDK.

Команды ChkINF имеют следующий общий формат:

```
ChkINF test.inf /B
```

Данная команда проверяет достоверность файла test.inf и выводит результаты в браузере по умолчанию (/B). Генерируемый отчет состоит из разделов с ошибками и предупреждениями, за которыми следует копия прокомментированного INF-файла, в котором показываются обнаруженные проблемы. Многие из предупреждающих сообщений указывают на проблемы, которые, в некоторых случаях, мешают драйверу загрузиться должным образом. Все проблемы, указанные инструментом ChkINF, необходимо исправить.

Инструмент ChkINF находится в папке %wdk%\tools\chkinf. Дополнительную информацию по инструменту ChkINF см. в разделе **ChkINF** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79776>.

## Инструмент Device Console

Инструмент командной строки Device Console (DevCon, Devcon.exe) выводит подробную информацию об устройствах. С помощью этого инструмента можно выполнять поиск устройств и манипулировать ими из командной строки. Инструмент DevCon разрешает, запрещает, устанавливает, конфигурирует и удаляет устройства на локальном компьютере и выводит подробную информацию об устройствах на локальном и удаленных компьютерах. Он может исполняться на машинах под управлением операционной системы Windows 2000 и более поздних версий Windows.

С помощью инструмента DevCon можно проверить правильность установки и конфигурации драйвера, включая правильность файлов INF, стека устройств, файлов драйвера и драйверного пакета. Команды инструмента DevCon — enable, disable, install, start, stop и continue — можно также организовывать в сценарии для проверки драйвера.

Набор разработчика WDK также содержит исходный код для этого инструмента в качестве примера, демонстрирующего применение интерфейса SetupAPI и функций установки устройства для перечисления устройств и выполнения операций с устройствами в консольном приложении.

Инструмент DevCon находится в папке %wdk%\tools\devcon, а его исходный код — в папке %wdk%\src\setup\devcon. Дополнительную информацию по инструменту DevCon см. в разделе **DevCon** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79777>.

## Инструмент Device Path Exerciser

Инструмент командной строки Device Path Exerciser (Dc2, Dc2.exe) применяется для проверки драйверов на надежность и безопасность. Он вызывает драйверы через различные интерфейсы ввода/вывода пользовательского режима с действительными, недействительными, бессмысленными и плохо сформированными буферами и данными, которые вызовут фатальный сбой драйвера, если их не обработать должным образом. С помощью этих тестов можно выявить некачественную конструкцию или реализацию драйвера, которые могут вызвать фатальный сбой системы или сделать ее уязвимой для злонамеренных действий.

### Полезная информация

При тестировании драйвера с помощью инструмента Device Path Exerciser также рекомендуется применять инструмент Driver Verifier. Таким образом, можно получить всесторонний профиль поведения драйвера под нагрузкой тестов инструмента Device Path Exerciser.

Инструмент Device Path Exerciser определяет драйверы, которые неправильно обрабатывают следующие вызовы:

- ◆ неожиданные запросы ввода/вывода к драйверу, такие как, например, запросы к файловой системе, направленные звуковой карте;
- ◆ запросы с буферами недостаточного размера для помещения всех возвращаемых данных;
- ◆ запросы IOCTL или запросы FSCTL (File-system control code, код управления файловой системой) без буферов, с буферами недостаточного размера или буферами, содержащими бессмысленную информацию;
- ◆ запросы IOCTL или FSCTL прямого (method Direct) или ни прямого, ни буферизованного (method Neither) доступа к данным, в которых данные изменяются асинхронно;
- ◆ запросы IOCTL или FSCTL типа method Neither с недействительными указателями;
- ◆ запросы IOCTL и FSCTL и запросы типа fast path query, в которых отображение пользовательского буфера меняется асинхронно, что делает страницы нечитабельными;
- ◆ операции открытия файла с произвольными или трудно поддающимися анализу именами относительного пути файла или операции открытия файла с несуществующими объектами устройств.

При тестировании инструмент Device Path Exerciser обычно посылает драйверу быстро следующие друг за другом сотни тысяч похожих вызовов. В этих вызовах применяются разные методы доступа к данным, действительные и недействительные адреса и размеры буферов, а также различные комбинации передавленных параметров функций, включая пробелы, строки и символы, которые могут быть неверно интерпретированы дефектной процедурой синтаксического анализа или обработки ошибки.

Инструмент Device Path Exerciser находится в папке установки набора WDK в подпапке %wdk%\tools\dc2. Дополнительную информацию об инструменте Device Path Exerciser см. в разделе **Device Path Exerciser** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79778>.

## Инструменты Kern Rate и Kern Rate Viewer

Инструмент KernRate — это профайлер, с помощью которого можно определить, каким образом расходуется процессорное время. Он также собирает статистические данные по не-

скольким элементам ядра, связанным с производительностью, таким как, например, элементы в утилите системного мониторинга (Sysmon). Инструмент Kern Rate Viewer (KRVView) отображает вывод инструмента KernRate в таблице Microsoft Excel.

Профилирование производительности драйвера следует выполнять двумя способами.

- ◆ Профилировать производительность всей системы в то время, как на ней исполняется ваш драйвер.

Даже при значительной нагрузке на драйвер процентное отношение времени, затраченное на исполнение драйвера, должно быть относительно низким.

- ◆ Профилировать производительность только самого драйвера.

При исследовании данных этого профиля ищите "горячие точки", в которых исполнение драйвера занимает много времени. В каждой из таких "горячих точек" ищите или проблему с кодом, или способ улучшить алгоритм, чтобы уменьшить нагрузку. Многие "горячие точки" отражают вполне нормальную работу драйвера, но при проверке в некоторых из них можно обнаружить ошибки, оказывающие отрицательный эффект на производительность.

Если вы никогда раньше не занимались профилированием производительности, имейте в виду, что, для того чтобы получить хорошие данные, необходимо выполнять их сбор в течение длительного времени. Чтобы получить солидные, надежные данные, следует выполнять профилирование в течение нескольких часов. Кроме этого, если драйвер не выполняет общераспространенные и представительные задачи, данные по производительности будут ненадежными. Поэтому необходимо организовать хороший тест, который может держать драйвер загруженным в течение длительного времени. Кроме времени, профилирование требует много памяти — профайлеры выборок считывают текущее значение счетчика команд и помещают данные в "корзину", которая указывает исполняемый код. Инструмент KernRate позволяет настраивать размер корзин; чем меньше размер корзины, тем более точная выборка.

Инструменты KernRate и KRVView можно загрузить с Web-сайта WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79779>.

## **Инструмент Plug and Play Driver Test**

Инструмент Plug and Play Driver Test (Pnptest, Pnptest.exe) нагружает различные ветви кода в драйвере и компонентах пользовательского режима. Чтобы получить наилучшие результаты, инструмент Plug and Play Driver Test следует применять совместно с инструментом Driver Verifier.

Инструмент Plug and Play Driver Test заставляет драйвер обрабатывать почти все пакеты IRP Plug and Play. Но наибольшее внимание он уделяет трем аспектам: удалению устройств, неожиданному удалению устройств и перераспределению ресурсов. В тестах применяется комбинация вызовов интерфейса API пользовательского режима в тестовом приложении и вызовов интерфейса API ядра через верхний драйвер фильтра. Каждый из этих аспектов можно тестировать самостоятельно или же тестировать их вместе в виде нагрузочного тестирования.

Инструмент Pnptest находится в папке %wdk%\tools\pnptest. Дополнительную информацию об инструменте Pnptest см. в разделе **Plug and Play Driver Test** (Инструмент Plug and Play Driver Test) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79780>.

## Инструмент Plug and Play CPU Test

Инструмент командной строки Plug and Play CPU Test (PNPCPU, Pnprcpu.exe) эмулирует подключение процессоров в исполняющийся экземпляр Windows Server Longhorn. Это действие иногда называется *горячим подключением*.

С помощью этого инструмента можно протестировать надежность работы драйвера или приложения при добавлении процессора в работающую систему. Если возможности драйвера или приложения были расширены и включают поддержку горячего подключения процессоров на системах, поддерживающих такое действие, инструмент можно использовать, чтобы удостовериться в том, что все релевантные извещения Plug and Play обрабатываются должным образом.

Инструмент PNPCPU находится в папке %wdk%\tools\pnpcpu. Дополнительную информацию по инструменту PNPCPU см. в разделе **PNPCPU** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79781>.

## Инструмент Memory Pool Monitor

Инструмент Memory Pool Monitor (PoolMon, Poolmon.exe) выводит на экран данные, собираемые операционной системой о выделениях памяти из системных страничного и нестраничного пулов памяти и из пулов памяти, используемых для сеансов Terminal Services. Данные группируются по тегу выделения пула.

Инструмент PoolMon можно использовать для выявления утечек памяти при создании нового драйвера, модифицировании кода существующего драйвера или при нагружочном тестировании драйвера. Его также можно применять на каждом этапе тестирования драйвера, чтобы увидеть закономерности выделений и освобождений памяти, а также объем памяти пула, используемой драйвером в любой момент времени.

Инструмент пользуется механизмом pool tagging<sup>1</sup> для вывода имен компонентов и драйверов Windows, которые назначают теги пулов, перечисленные в файле Pooltag.txt. (Этот файл устанавливается с инструментом PoolMon и пакетом Debugging Tools for Windows.) Чтобы использовать инструмент PoolMon на системах с Windows XP или более ранними версиями Windows, необходимо разрешить pool tagging. В системах под управлением Windows Server 2003 и более поздних версий Windows pool tagging разрешено на постоянной основе.

Инструмент PoolMon находится в папке %wdk%\tools\other. Дополнительную информацию по инструменту PoolMon см. в разделе **PoolMon** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=7978>.

## Инструмент Power Management Test Tool

Инструмент Power Management Test Tool (PwrTest, PwrTest.exe) осуществляет управление энергопотреблением и получает от системы и записывает информацию о событиях управления энергопотреблением. С помощью инструмента PwrTest можно автоматизировать переходы в состояние сна и пробуждения, а также протоколировать информацию об управлении энергопотреблением процессором и информацию о состоянии аккумуляторной батареи на протяжении определенного периода времени.

<sup>1</sup> Тег пула (pool tag) — это тег размером от одного до четырех символов, назначаемый каждому выделению памяти пула. Назначения этих тегов по-английски называется "pool tagging", что дословно переводится как "пуловое тегирование", а одним из вариантов литературного перевода может быть описание процесса — назначение тега выделенной памяти из пула. — *Пер.*

Инструмент PwrTest находится в папке %wdk%\tools\acpi\pwrtest. Дополнительную информацию по этому инструменту см. в разделе **PwrTest** в WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79783>.

Особыми свойствами инструмента PwrTest являются надежное протоколирование и интерфейс командной строки. В табл. 21.1 приводится список сценариев управления энергопотребления, которые можно протестировать с помощью инструмента PwrTest.

**Таблица 21.1. Сценарии управления энергопотребления, тестируемые инструментом PwrTest**

Сценарий	Исполняемые действия
SLEEP	Задействует функциональность перехода в состояние сна и пробуждения. Переходы в состояние сна и пробуждения можно автоматизировать; также можно указывать требуемые состояния сна
PPM	Выводит и записывает информацию и метрики управления энергопотреблением процессора. Информация о производительности (состояние ACPI P) и бездействии процессора (состояние ACPI C) можно протоколировать в течение периода времени через указанные временные интервалы
BATTERY	Выводит и протоколирует информацию и метрики аккумуляторной батареи. Такие показатели, как емкость аккумулятора, напряжение, скорость разрядки и расчетный оставшийся заряд можно протоколировать в течение периода времени через указанные временные интервалы
INFO	Выводит на экран информацию об управлении энергопотреблением, такую как доступные состояния сна и возможности управления энергопотреблением процессора
ES	Выводит на экран и протоколирует изменения рабочих состояний потоков. Изменение рабочих состояний потоков позволяет приложениям и сервисам временно подавлять настройки управления энергопотреблением системы, такие как значение тайм-аута для выключения монитора и значение тайм-аута для перевода системы в состояние сна

## Инфраструктура WDTF

Инфраструктура WDTF (Windows Device Testing Framework, инфраструктура тестирования устройств Windows) предоставляет набор интерфейсов COM, с помощью которых можно создавать собственные сценарии тестов для устройств. Инфраструктура WDTF упрощает несколько базовых сценариев тестирования устройств.

### ◆ Нахождение подключенных к компьютеру устройств.

Нахождение устройств можно выполнять по технологии (например, USB или Bluetooth), по классу устройства (например, устройство хранения) или по другим критериям. Можно также идентифицировать определенное устройство по имени или выполнять поиск устройств по составному критерию. Кроме этого, можно выполнять поиск устройств, связанных друг с другом отношениями типа "родитель — потомок" или графовыми отношениями.

### ◆ Создание коллекций тестируемых объектов.

Эта функциональность полезна в тех случаях, когда нужно протестировать устройства, имеющие общие характеристики. Например, можно запросить коллекцию всех дисковых устройств, после чего протестировать всю коллекцию группой.

◆ **Выполнение операций на одном или нескольких устройствах.**

Можно создать общий сценарий для тестирования набора устройств, даже если индивидуальные устройства в наборе разных типов.

◆ **Исполнение простых тестов функциональностей на одном или нескольких устройствах, не зная базового типа устройства.**

Можно выполнять простые синхронные операции ввода/вывода на устройстве с помощью рабочего интерфейса устройства SimpleIO. Интерфейс SimpleIO абстрагирует операции ввода/вывода, позволяя использовать стандартный интерфейс для инициирования тестирования на каждом устройстве, даже если подлежащий код, выполняющий фактические операции тестирования, разный для каждого устройства. Инфраструктура содержит библиотеку реализаций интерфейса SimpleIO и определяет и создает экземпляр необходимого интерфейса для каждого типа устройств.

◆ **Запускать, приостанавливать и останавливать нагрузочное тестирование.**

Можно асинхронно выполнять простые операции ввода/вывода на устройстве и управлять тестированием во время исполнения с помощью интерфейса SimpleIOTstress. Интерфейс SimpleIOTstress реализован в виде оберточного интерфейса для интерфейса SimpleIO.

◆ **Использовать код, реализующий тестирование, не имея специфических знаний об устройстве.**

Пользователи могут писать сценарии WDTF, в которых используется код, специфический для устройств и классов устройств, не понимая, каким образом каждое устройство реализует используемые ими интерфейсы.

Тестовые сценарии WDTF можно писать на любом языке программирования, который поддерживает автоматизацию COM. Набор разработчика драйверов WDK содержит следующие образцы сценариев (табл. 21.2), с помощью которых можно выполнять тестирование драйверов KMDF и UMDF.

**Таблица 21.2. Образцы сценариев в наборе разработчика драйверов WDK**

Сценарий	Выполняемый тест
Disable_Enable_With_IO.wsf	Запрещает и разрешает по одному все устройства, которые можно запретить. Этот образец также выполняет проверку ввода/вывода на всех устройствах, связанных с выбранными устройствами
Sleep_Stress_With_IO.wsf	Подвергает устройства нагрузочному тестированию управления энергопотреблением
Common_Scenario_Stress_With_IO.wsf	Подвергает устройства нагрузочному тестированию Plug and Play и управления энергопотреблением
Basic_SimpleIO.wsf	Выполняет часть "With_IO" стандартных сценариев, описанных для предыдущих сценариев
EnumDevices.wsf	Выводит информацию об устройствах, обнаруженных на компьютере

Библиотеки и образцы сценариев WDTF находятся в установочной папке WDK в подпапке %wdk%\tools\WDTF. Дополнительную информацию по этому вопросу см. в разделе Windows Device Testing Framework на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79785>.

### Полезная информация

Инфраструктура WDTF устанавливается автоматически средством тестирования Driver Test Manager (DTM), поставляемым в наборе WDK для выполнения тестов для программы Windows Logo Program. Поэтому если вы используете средство DTM, то вам не нужно отдельно устанавливать инфраструктуру WDTF.

## Инструмент Driver Verifier

Инструмент Driver Verifier является чрезвычайно полезным для выявления ошибок в драйверах режима ядра. Он тестирует и улавливает многие ситуации, которые иначе были бы незамечены при нормальной работе. Инструмент Driver Verifier проверяет, чтобы драйверы не вызывали запрещенных функций или не вызывали искажений системного кода. Driver Verifier устанавливается со всеми версиями Windows Vista, Windows Server 2003, Windows XP и Windows 2000.

### Полезная информация

Все драйверы режима ядра, разрабатываемые корпорацией Microsoft, должны успешно пройти проверку инструментом Driver Verifier со стандартными настройками. Разработчики корпорации Microsoft рекомендуют активировать Driver Verifier как можно раньше и пользоваться им на протяжении всего времени разработки драйвера.

Инструмент Driver Verifier эволюционировал со временем, поэтому версии Driver Verifier, установленные в более поздних выпусках Windows имеют больше возможностей, чем более ранние версии. Одним из значительных улучшений инструмента Driver Verifier, поставляемого с Windows Vista, является возможность активировать большинство опций и добавлять и удалять драйверы для верификации, не перезагружая компьютер.

В зависимости от выбранных опций и количества проверяемых драйверов, исполнение Driver Verifier может отрицательно сказаться на производительности системы. Поэтому при тестировании драйверов по аспектам производительности обязательно отключите Driver Verifier, чтобы он не показал результаты.

### Примечание

Driver Verifier проверяет достоверность вызовов функций WDM режима ядра. При проверке драйвера WDF, используя Driver Verifier, выводимая информация может содержать имена функций, которые драйвер не вызывает явно, но которые вызываются для него инфраструктурой WDF. Имена этих базовых функций WDM приводятся в этом разделе при обсуждении работы Driver Verifier.

Дополнительную информацию по Driver Verifier см. в одноименном разделе на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79793>. Также см. раздел **Driver Verifier in Windows Vista** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79588>.

## Когда использовать Driver Verifier

Инструмент Driver Verifier может использоваться на протяжении всего процесса разработки и тестирования драйвера, чтобы помочь найти проблемы на ранних стадиях разработки, когда их легче и дешевле исправить. С его помощью можно выявить такие состояния, как нарушение целостности памяти, неправильно обрабатываемые пакеты IRP, некорректное ис-

пользование буферов DMA и возможные взаимоблокировки. Так как Driver Verifier устанавливается вместе с Windows, его можно применять при поиске и устранении проблем на компьютерах пользователей.

## Как работает Driver Verifier

Driver Verifier работает с драйверами, установленными и исполняющимися в системе. Для тестирования поведения драйвера в предельных условиях инструмент перехватывает определенные системные вызовы драйвера и вставляет ошибки или ограничивает ресурсы.

В своей работе Driver Verifier всегда проверяет использование драйвером памяти на правильном уровне IRQL, правильность получения и освобождения спин-блокировок, правильность выделения и освобождении памяти, а также удаление таймеров, прежде чем освобождать память, выделенную из пула. При выгрузке драйвера Driver Verifier всегда проверяет, освободил ли драйвер свои ресурсы должным образом.

Кроме только что перечисленных, Driver Verifier может выполнять дополнительные проверки. В табл. 21.3 перечислены опции, которые можно указать для дополнительных проверок.

**Таблица 21.3. Опции Driver Verifier**

Опция	Действие Driver Verifier
<b>Deadlock Detection</b> (Выявление взаимоблокировок)	Отслеживает использование драйвером спин-блокировок, мьютексов и быстрых мьютексов, чтобы определить, может ли код драйвера потенциально вызвать взаимоблокировку. (Применимо для Windows XP и более поздних версий.)
<b>Disk Integrity Checking</b> (Проверка целостности данных диска)	Отслеживает обращения к жесткому диску и определяет, предохраняет ли диск свои данные должным образом. (Применимо для Windows Server 2003 и более поздних версий.)
<b>DMA Verification</b> (Проверка DMA)	Отслеживает использование драйвером процедур DMA, чтобы выявить некорректное использование буферов, адаптеров и регистров отображения DMA. (Применимо для Windows XP и более поздних версий.)
<b>Driver Hang Verification</b> (Проверка драйвера на зависание)	Замеряет время исполнения процедур драйвера для завершения и отмены запросов ввода/вывода и извещает о процедурах, время исполнение которых превышает указанное время. (Применимо для Windows Vista и более поздних версий.)
<b>Enhanced I/O Verification</b> (Расширенная проверка ввода/вывода)	Отслеживает вызовы нескольких процедур менеджера ввода/вывода и выполняет нагружочное тестирование пакетов IRP Plug and Play, пакетов IRP управления энергопотреблением и пакетов IRP интерфейса WMI. (Применимо для Windows XP и более поздних версий.)
<b>Force IRQL Checking</b> (Принудительная проверка уровня IRQL)	Подвергает драйвер экстремальной нагрузке, делая недействительным страничный код. Выявляет попытки драйвера обратиться к страничной памяти на неправильном уровне IRQL или когда он удерживает спин-блокировку
<b>Force Pending I/O Requests</b> (Принудительное присвоение запросам ввода/вывода статуса ожидания исполнения)	Проверяет реакцию драйвера на возвращенное значение статуса STATUS_PENDING, возвращая значение STATUS_PENDING для произвольных вызовов процедуры IoCallDriver. (Применимо для Windows Vista и более поздних версий.)

Таблица 21.3 (окончание)

Опция	Действие Driver Verifier
I/O Verification (Проверка ввода/вывода)	Выделяет пакеты IRP драйвера из специального пула и отслеживает их обработку драйвером, чтобы выявить некорректное или несоответствующее использование процедур ввода/вывода. В Windows Vista и более поздних версиях Windows включение проверки ввода/вывода также включает проверку драйвера на зависание (Driver Hang Verification)
IRP Logging (Протоколирование пакетов IRP)	Отслеживает и протоколирует использование драйвером пакетов IRP. (Применимо для Windows Server 2003 и более поздних версий.)
Low Resources Simulation (Эмуляция низкого уровня ресурсов)	В произвольном порядке вызывает неудачное завершение запросов на выделение памяти и запросов на выделение других ресурсов. Внедрением этих ошибок выделения ресурсов Driver Verifier проверяет способность драйвера справляться с ситуациями нехватки ресурсов
Miscellaneous Checks (Прочие проверки)	Ищет распространенные причины фатальных сбоев драйверов, такие как, например, неправильная обработка освобожденной памяти. (Применимо для Windows Vista и более поздних версий.)
Pool Tracking (Отслеживание пулов)	Проверяет, освобождает ли драйвер все выделения памяти при выгрузке, таким образом выявляя утечки памяти
Special Pool (Специальный пул)	Выделяет память для большей части запросов драйвера на выделение памяти из специального пула, который отслеживается на предмет попыток доступа к памяти после конца (memory overrun) и до начала (memory underrun) выделенной области, а также на предмет попыток доступа к освобожденной памяти

## Как работать с Driver Verifier

При запуске Driver Verifier указываются драйвер или драйверы, которые нужно проверить, и требуемые для проверки опции. После этого Driver Verifier отслеживает выбранные драйверы до тех пор, пока он не деактивирован или до перезагрузки системы. Вывод из Driver Verifier можно направить в файл журнала. Также, если к системе подключен отладчик, можно применить расширение отладчика !verifier, чтобы просматривать выводимую Driver Verifier информацию в отладчике.

Driver Verifier активируется и отслеживается утилитой Verifier (Verifier.exe), которая находится в папке %windir%\system.

## Использование утилиты Verifier из командной строки

Введите команду verifier с одним параметром, по крайней мере.

Например, следующая команда активирует Driver Verifier с опцией отслеживания пулов (/flags 8) для драйвера xyz.sys:

```
Verifier /flags 8 /driver xyz.sys
```

Полную информацию об опциях командной строки для Driver Verifier см. в разделе **The Verifier Command Line** (Командная строка утилиты Verifier) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79788>.

## Запуск Driver Verifier Manager

Введите команду `Verifier` без параметров. Эта команда откроет начальное окно Driver Verifier Manager с простым графическим пользовательским интерфейсом.

В последующих окнах предоставляются опции выбора драйверов и стандартных или индивидуальных настроек для их проверки. Потом Driver Verifier Manager запускает утилиту Verifier с выбранными настройками.

Дополнительную информацию на эту тему см. в разделе **Driver Verifier Manager (Windows XP and Later)** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79789>.

## Примеры работы с Driver Verifier

Далее приводится несколько примеров команд утилиты Verifier, а также типичные параметры, используемые в них. На практике следует сочетать параметры таким образом, чтобы активировать проверки, требующиеся для данного драйвера и сценария его тестирования.

### Пример 1: активирование стандартных опций для нескольких драйверов

Следующая команда активирует набор стандартных опций инструмента Driver Verifier для указанных драйверов. Опции вступают в действие после перезагрузки системы.

```
Verifier/standard/drivers список_драйверов
```

В Windows XP и более поздних версиях Windows параметр `/standard` активирует следующие опции:

- ◆ **Deadlock Detection** (Выявление взаимоблокировок);
- ◆ **I/O Verification** (Проверка ввода/вывода);
- ◆ **DMA Verification** (Проверка DMA);
- ◆ **Pool Tracking** (Отслеживание пулов);
- ◆ **Force IRQL Checking** (Принудительная проверка уровня IRQL);
- ◆ **Special Pool** (Специальный пул).

В Windows Vista и более поздних версиях Windows опции `/standard` также включают **Security checks** (Проверки безопасности) и **Miscellaneous Checks** (Прочие проверки).

Параметр `/drivers` указывает Driver Verifier выполнять проверку указанных драйверов. Драйверы для проверки указываются в параметре `список_драйверов` по имени двоичного файла драйвера, например, `Driver.sys`. Имена драйверов разделяются пробелом.

### Пример 2: активирование специальных опций для всех драйверов

Следующая команда задает одну или несколько опций для всех драйверов компьютерной системы:

```
Verifier	flags опции /all
```

Параметр `/flags` задает активирование указанных опций после перезагрузки системы. Сами опции указываются в параметре `опции` комбинацией значений (в десятичном или шестнадцатеричном формате), перечисленных в табл. 21.4.

**Таблица 21.4.** Опции параметра /flags инструмента Driver Verifier

Десятичный формат	Шестнадцатеричный формат	Опция
1	0x1	<b>Special Pool</b> (Специальный пул)
2	0x2	<b>Force IRQL Checking</b> (Принудительная проверка уровня IRQL)
4	0x4	<b>Low Resources Simulation</b> (Эмуляция низкого уровня ресурсов)
8	0x8	<b>Pool Tracking</b> (Отслеживание пулов)
16	0x10	<b>I/O Verification</b> (Проверка ввода/вывода)
32	0x20	<b>Deadlock Detection</b> (Выявление взаимоблокировок). Применимо для Windows XP и более поздних версий
64	0x40	<b>Enhanced I/O Verification</b> (Расширенная проверка ввода/вывода). Применимо для Windows XP и более поздних версий
128	0x80	<b>DMA Verification</b> (Проверка DMA). Применимо для Windows XP и более поздних версий
256	0x100	<b>Security checks</b> (Проверки безопасности). Применимо для Windows XP и более поздних версий
512	0x200	<b>Force Pending I/O Requests</b> (Принудительное присвоение запросам ввода/вывода статуса ожидания исполнения). Применимо для Windows Vista и более поздних версий
1024	0x400	<b>IRP Logging</b> (Протоколирование пакетов IRP). Применимо для Windows Server 2003 и более поздних версий
2048	0x800	<b>Miscellaneous Checks</b> (Прочие проверки). Применимо для Windows Vista и более поздних версий

Например, в Windows Vista параметр /standard, показанный в примере 1, является эквивалентом /flags 0x9BB.

### Пример 3: запуск и остановка проверки драйвера без перезагрузки

Чтобы начать проверку любого драйвера без перезагрузки системы, даже если драйвер уже загружен, применяется следующий синтаксис:

```
verifier /volatile /adddriver DriverName.sys
```

Параметр /volatile изменяет настройки, которые вступают в действие немедленно, без перезагрузки компьютера.

Чтобы удалить драйвер со списка драйверов для проверки, применяется параметр /removedriver, как показано в следующем примере:

```
verifier /volatile /removedriver DriverName.sys
```

Параметр /removedriver удаляет драйвер из списка драйверов для проверки, только если драйвер еще не был загружен. Если драйвер уже был загружен, то Driver Verifier продолжает отслеживать этот драйвер до следующей перезагрузки системы. Чтобы свести к минимуму

му накладные расходы до следующей перезагрузки, следует деактивировать все опции Driver Verifier, как показано в примере 5.

### Пример 4: активирование или деактивирование опций без перезагрузки

Для активирования и деактивирования любой опции без перезагрузки системы применяется параметр `/volatile` с необходимыми опциями, указанными в параметре `/flags`, как показано в следующем примере:

```
verifier /volatile /flags [опции]
```

Этот синтаксис можно использовать для любой опции Driver Verifier, за исключением опций **SCSI Verification** (Проверка SCSI) и **Disk Integrity Checking** (Проверка целостности диска). Например, следующая команда активирует опцию **Deadlock Detection** (Выявление взаимоблокировок), не требуя перезагрузки системы:

```
verifier /volatile /flags 0x20
```

### Пример 5: деактивирование всех опций Driver Verifier

Для деактивирования всех опций инструмента Driver Verifier без перезагрузки системы применяются параметры `/volatile` и `/flags`, как показано в следующем примере:

```
verifier /volatile /flags 0
```

Driver Verifier продолжает отслеживать драйвер с опциями автоматической проверки, которую нельзя отключить. Но накладные расходы автоматических проверок составляют только около 10% от накладных расходов типичной проверки, поэтому они оказывают сравнительно небольшой эффект на производительность.

### Пример 6: деактивирование Driver Verifier

Для очистки всех настроек инструмента Driver Verifier применяется команда:

```
verifier /reset
```

После следующей перезагрузки проверка драйверов не будет выполняться.

### Пример 7: эмуляция недостаточных ресурсов

Когда активирована опция **Low Resources Simulation** (Эмуляция низкого уровня ресурсов), Driver Verifier вызывает неудачное завершение произвольных выделений памяти, эмулируя ситуацию исполнения драйвера на компьютере с ограниченным объемом памяти. Таким образом, проверяется способность драйвера реагировать на ситуации низкого уровня памяти или других ресурсов.

Проверка **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) вызывает неудачное завершение запросов выделения памяти, выдаваемых вызовами нескольких разных функций, включая следующие функции:

- ◆ `ExAllocatePoolXxx;`
- ◆ `MmMapIoSpace;`

- ◆ MmMapLockedPagesSpecifyCache;
- ◆ MmProbeAndLockPages.

В Windows Vista и более поздних версиях Windows неудачное завершение произвольных запросов на выделение памяти также осуществляется в вызовах следующих функций:

- ◆ IoAllocateErrorLogEntry;
- ◆ IoAllocateIrp;
- ◆ IoAllocateMdl;
- ◆ IoAllocateWorkItem;
- ◆ MmAllocateContiguousMemoryXxx;
- ◆ MmAllocatePagesForMdl.

Опция **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) включается следующей командой:

```
Verifier /faults [Probability PoolTags Applications DelayMins] /driver DriverList
```

Параметры задаваемых настроек (Probability, PoolTags и т. д.) должны перечисляться в показанном порядке. Если нужно опустить какое-либо значение, его место заполняется кавычками.

Далее приводится объяснение параметров, применяемых для опции **Low Resources Simulation** (Эмуляция низкого уровня ресурсов).

◆ Параметр **Probability**.

Задает вероятность неудачного завершения инструментом Driver Verifier данного выделения памяти. Вероятность задается в виде числа (в десятичном или шестнадцатеричном формате) выделений памяти, которые Driver Verifier завершит неудачей на каждые 10 000 выделений памяти. По умолчанию применяется значение 600, что означает 600/10000, или 6%.

◆ Параметр **PoolTags**.

Задает теги пулов, для которых Driver Verifier может завершить неудачей выделение памяти. По умолчанию неудачей могут завершаться все выделения памяти. Для указания нескольких тегов пулов можно применять символ подстановки \*. Например, abc\*. При указании нескольких тегов пулов теги разделяются пробелами, а список заключается в кавычки.

◆ Параметр **Applications**.

Задает приложения, для которых Driver Verifier может завершить неудачей выделение памяти. Параметр **Applications** содержит имена одного или нескольких исполняемых файлов. По умолчанию неудачей могут завершаться выделения памяти для всех приложений. В списке имена программ разделяются пробелами, а список заключается в кавычки.

◆ Параметр **DelayMins**.

Задает период времени в минутах после загрузки, на протяжении которого Driver Verifier не вызывает преднамеренно никаких неудач выделений памяти. Эта задержка позволяет драйверам загрузиться, а системе стабилизироваться, перед тем, как начать тестирование. Значение параметра **DelayMins** указывается в десятичном или шестнадцатеричном формате. Значение по умолчанию — 7.

Представленная далее команда включает опцию **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) с вероятностью отказа в 10% (1000/10000) и задержкой в 5 минут для тегов пулов Tag1 и Fred и приложения Notepad.exe:

```
Verifier /faults 1000 "Tag1 Fred" Notepad.exe 5
```

А следующая команда включает опцию **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) со стандартными значениями параметров, но увеличивает задержку до 10 минут:

```
Verifier /faults "" "" "" 0xa
```

Параметр **/volatile** можно применять для изменения настроек опции **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) без перезагрузки компьютера, как показано в следующем примере:

```
Verifier /volatile /faults [Probability PoolTags Applications DelayMins /driver Driverlist]
```

Эти настройки вступают в действие немедленно.

### Пример 8: принудительная проверка уровня IRQL (Force IRQL Checking)

Хотя драйверам режима ядра нельзя обращаться к страничной памяти на высоком уровне IRQL или во время удерживания спин-блокировки, такое действие не обязательно вызовет ошибку, если данная страница памяти не была в действительности удалена из рабочего множества и вытеснена на диск.

При включенной опции **Force IRQL Checking** (Принудительная проверка уровня IRQL) Driver Verifier полностью удаляет страничный код и данные драйвера из рабочего множества при каждой попытке драйвера получить спин-блокировку, вызвать процедуру KeSynchronizeExecution или повысить уровень IRQL до значения DISRATCH\_LEVEL или выше. Если драйвер попытается обратиться к любой области такой страничной памяти, Driver Verifier вызывает останов bugcheck.

В Windows Vista инструмент Driver Verifier может выявить выделение определенных объектов синхронизации в страничной памяти. Под эти объекты синхронизации не должна выделяться страничная память, т. к. существует возможность обращения к ним ядра на высоком уровне IRQL. Driver Verifier может обнаружить выделение страничной памяти для следующих объектов:

- ◆ ERESOURCE;
- ◆ FAST\_MUTEX;
- ◆ KEVENT;
- ◆ KMUTEX;
- ◆ KSEMAPHORE;
- ◆ KSPIN\_LOCK;
- ◆ KTIMER.

Опция **Force IRQL Checking** (Принудительная проверка уровня IRQL) включена в стандартные настройки **/standard**. В индивидуальном порядке ее можно включить следующей командой:

```
Verifier /flags 0x2 /driver DriverList
```

Опция вступает в действие после перезагрузки. Для включения опции **Force IRQL Checking** (Принудительная проверка уровня IRQL) без перезагрузки следует применять параметр /volatile.

## Использование информации от Driver Verifier при отладке

При отладке статистические данные, связанные с Driver Verifier, можно отслеживать и выводить с помощью расширения отладчика !verifier. В этом разделе приводится пример, демонстрирующий применение расширения !verifier для отладки фатального сбоя, вызванного опцией **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) инструмента Driver Verifier.

Предоставленная информация действительна при использовании этой опции на всех версиях Windows.

### Пример 1: просмотр трассировок операций стека с помощью !verifier

Наверное, наиболее легко поддаются пониманию фатальные сбои, вызванные обращением драйвера к указателю `NULL`. В данном примере анализ исходного кода в области ветви исполнения, вызвавшей фатальный сбой, показывает, что драйвер вызвал функцию `ExAllocatePoolWithTag`, эта функция возвратила значение `NULL`, а драйвер не проверил возвращенное значение и поэтому при использовании этого указателя потерпел фатальный сбой.

Но понять причины, вызвавшие фатальный сбой драйвера, не всегда так просто. Часто полезную информацию можно получить, анализируя трассировки стека для недавних внедрений ошибок. Например, команда `!verifier` в отладчике ядра выводит четыре, наиболее недавно внесенные ошибки, но необязательно четыре трассировки стека. Большее число трассировок стека можно вывести, указав дополнительный параметр для команды `!verifier`. Трассировка стека для самой недавней ошибки выводится первой.

Если при включенном внесении ошибок происходит сбой системы или переход в отладчик, то можно использовать трассировку стека, чтобы определить, не было ли это вызвано внесенной ошибкой. Например, если отладчик указывает сбой в `win32k!GreEnableEUDC`, то из листинга 21.1 можно определить, что сбой был вызван последней внесенной ошибкой.

#### Листинг 21.1. Пример сбоя в `win32k!GreEnableEUDC`

```
kd> !verifier 4
Resource fault injection history:
Tracker @ 8354A000 (# entries: 80, size: 80, depth: 8)
Entry @ 8354B258 (index 75)
    Thread: C2638220
    816760CB nt!VerifierExAllocatePoolWithTag+0x49
    A4720443 win32k!bDeleteAllF1Entry+0x15d
    A4720AB0 win32k!CreEnableEUDC+0x70
    A47218FA win32k!CleanUpEUDC+0x37
    A473998E win32k!GdiMultiUserFontCleanup+0x5
    815AEACC nt!MiDereferenceSession+0x74
    8146D3B4 nt!MmCleanProcessAddressSpace+0x112
    815DF739 nt!PspExitThread+0x603
```

```

Entry @ 8354B230 (index 74)
  Thread: 8436D770
  816760CB nt!VerifierExAllocatePoolWithTag+0x49
  A462141C win32k!Win32AllocPool+0x13
  A4725F94 win32k!StubGdiAlloc+0x10
  A4631A93 win32k!ExAllocateFromPagedLookasideList+0x27
  A47261A4 win32k!AllocateObject+0x23
  A4726F76 win32k!HmgAlloc+0x25
  A47509D8 win32k!DCMEMOBJ::DCMEMOBJ+0x3b
  A4717D61 win32k!CreCreateDisplayDC+0x31
Entry @ 8354B208 (index 73)
  Thread: D6B4B9B8
  816760CB nt!VerifierExAllocatePoolWithTag+0x49
  A462141C win32k!Win32AllocPool+0x13
  A46C2759 win32k!PALLOCMEM+0x17
  A477CCF2 win32k!bComputeCISET+0x82
  A477D07D win32k!PFEMEMOBJ::bInit+0x248
  A475DC18 win32k!PFFMEMOBJ::bAddEntry+0x6c
  A475E3E6 win32k!PFFMEMOBJ::bLoadFontFileTable+0x81
  A475BADE win32k!PUBLIC_PFTOBJ::bLoadFonts+0x2c4
Entry @ 8354B1E0 (index 72)
  Thread: CCAOA480
  816760CB nt!VerifierExAllocatePoolWithTag+0x49
  813B8C30 fltmgr!FltpAllocateIrpCtrl+0x122
  813CB1C9 fltmgr!FltpCreate+0x28d
  81675275 nt!IovCallDriver+0x1b1
  8141EDF1 nt!IofCallDriver+0x1f
  81566106 nt!IopParseDevice+0xde6
  815B9916 nt!ObpLookupObjectName+0x61a
  815B72D5 nt!ObOpenObjectByName+0xf7

```

Наиболее недавняя ошибка выделения была вызвана в пути кода `GreEnableEUDC`. Как мы помним, `GreEnableEUDC` — это функция, в которой произошел сбой в этом примере. Обратите внимание, что ошибка выделения произошла в контексте потока `C2638220`. Если при выполнении `!thread -1` адрес текущего потока будет `C2638220`, тогда вероятность еще больше, что наиболее недавно внесенная ошибка связана с недавним сбоем. Необходимо просмотреть исходный код в этой области, в поисках ветви кода, которая могла вызвать этот вид сбоя.

Часто текущий сбой связан с наиболее недавно внесенной ошибкой. Если анализ наиболее недавней трассировки стека не приносит результатов, то следует проанализировать другие трассировки стеков. Если и это окажется безрезультатным, можно выполнить команду `!Verifier 4 80`, чтобы вывести последние `0x80` трассировок стека, одна из которых может оказаться полезной.

## **Пример 2: использование `!Verifier` для вывода счетчиков ошибок и выделений памяти из пула**

Driver Verifier ведет счет числа ошибок, внесенных после последней перезагрузки. Также ведется счет числа попыток выделения памяти из пула. Значения этих двух счетчиков могут быть полезными при выяснении следующих ситуаций.

- ◆ В действительности ли опция **Low Resources Simulation** (Эмуляция низкого уровня ресурсов) вносит ошибки в тестируемый драйвер.

Например, если драйвер не выделяет памяти из пула, то ошибки не вносятся.

- ◆ Не было ли внесено слишком много ошибок.

Например, если число внесенных ошибок слишком большое по сравнению с числом попыток выделения памяти, то для следующего прохода теста вероятность внесения ошибок можно уменьшить.

- ◆ Не было ли число внесенных ошибок слишком низким.

В этом случае для следующих проходов тестирования необходимо увеличить значение вероятности внесения ошибок.

Счетчики выделений памяти можно вывести на экран с помощью команды `!Verifier` (листинг 21.2).

#### Листинг 21.2. Пример вывода счетчиков выделений памяти из пула

```
! verifier
Verify Level 5 ... enabled options are:
    Special pool
    Inject random low-resource API failures
Summary of All Verifier Statistics
RaiseIrqls                      0x2c671f
AcquireSpinLocks                 0xc1a02
Synch Executions                  0x10a623
Trims                            0x0
Pool Allocations Attempted      0x862e0e
Pool Allocations Succeeded       0x8626e3
Pool Allocations Succeeded SpecialPool 0x768060
Pool Allocations With NO TAG    0x0
Pool Allocations Failed          0x34f
Resource Allocations Failed Deliberately 0x3f5
```

## Инструмент KMDF Verifier

KMDF содержит встроенный верификатор, чьи возможности дополняют возможности Driver Verifier и позволяют выполнять дополнительные проверки, специфические для KMDF. Инструмент KMDF Verifier (иногда также называемый frameworks verifier) предоставляет исчерпывающие сообщения трассировки, предоставляющие подробную информацию о процессах, протекающих внутри инфраструктуры. Он отслеживает обращения ко всем объектам KMDF и создает журнал трассировки, который можно направить отладчику.

Подробную информацию об использовании верификатора KMDF Verifier см. в разделе **Debugging a Framework-based Driver** (Отладка драйверов KMDF) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79790>.

### Какая разница между Driver Verifier и KMDF Verifier?

Верификатор Driver Verifier — это верификатор общего типа, который проверяет, отвечает ли любой драйвер, включая драйверы KMDF, требованиям правил WDM. А верификатор KMDF Verifier проверяет следование драйверами KMDF более специфичным правилам KMDF. Кроме этого, верификатор KMDF Verifier предоставляет механизм последовательного внесения ошибок, которого нет в Driver Verifier. В Windows Vista верификатор Driver Verifier позволяет указывать вероятность сбоя, а KMDF Verifier может вызывать неудачное завершение каждой попытки выделения последовательно, начиная с энного выделения.

## Когда использовать KMDF Verifier

Во время разработки драйвера следует использовать как Driver Verifier, так и KMDF Verifier. При тестировании KMDF Verifier нужно активировать перед загрузкой тестируемого драйвера. Перезагружать систему после активирования KMDF Verifier не требуется.

## Как работает KMDF Verifier

KMDF Verifier работает с установленными и исполняющимися драйверами. Он предоставляет большое количество сообщений трассировки, которые выдают подробную информацию о процессах, протекающих внутри инфраструктуры. KMDF Verifier отслеживает обращения ко всем объектам KMDF и создает трассировку, которую можно направить отладчику.

KMDF Verifier выполняет следующие операции:

- ◆ проверяет захваты и иерархию блокировок;
- ◆ проверяет, что вызовы к инфраструктуре выполняются на правильном уровне IRQL;
- ◆ проверяет, что отмена ввода/вывода и использование очередей выполняются правильным образом;
- ◆ обеспечивает, что драйвер и инфраструктура следуют задокументированным контрактам.

KMDF Verifier может также эмулировать состояние низкого уровня и нехватки памяти. Он проверяет реакцию драйвера на эти ситуации, чтобы определить, может ли драйвер реагировать на них должным образом, т. е. без сбоя, зависания или неудачного завершения выгрузки.

## Как активировать KMDF Verifier

По умолчанию KMDF Verifier отключен, т. к. его экстенсивные проверки могут отрицательно сказаться на производительности системы.

Чтобы активировать KMDF Verifier:

1. Если драйвер уже загружен, с помощью Диспетчера устройств отключите его устройство. Отключение устройства вызовет выгрузку драйвера.
2. Запустите редактор реестра RegEdit и присвойте ненулевое значение параметру `VerifierOn` в подразделе `Parameters\Wdf` раздела `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Service` реестра Windows.

Ненулевое значение этого параметра означает, что KMDF Verifier активирован:

```
VerifierOn      REG_DWORD      0x1
```

3. С помощью Диспетчера устройств включите устройство, таким образом загружая драйвер.

Указание значения `VerifierOn` неявно устанавливает значение другого параметра реестра, `VerifyOn`, которое включает макрос `WDFVERIFY`, определенный в заголовочном файле `Wdfassert.h`. Макрос `WDFVERIFY` можно использовать в своем коде для проверки логических выражений. Если значение выражения равняется `FALSE`, то исполнение драйвера прерывается вызовом отладчика. Чтобы отключить этот макрос, присвойте этому параметру нулевое значение.

Для отключения KMDF Verifier выполняется та же последовательность шагов, что и для его активирования, только параметру `VerifierOn` присваивается нулевое значение.

Когда KMDF Verifier активирован, то путем манипулирования значениями параметров реестра можно включать следующие опции.

- ◆ **Эмуляция низкого уровня памяти.** Чтобы включить эмуляцию низкого уровня памяти, в редакторе реестра нужно присвоить значение  $n > 0$  параметру `VerifierAllocateFailCount` подраздела `Parameters\Wdf` драйвера в реестре Windows. После *n* попыток выделить память инфраструктура будет завершать неудачей все последующие попытки. Эти неуспешные выделения помогают проверить способность драйвера работать в условиях низкого уровня памяти.
- ◆ **Прерывание исполнения драйвера выходом в отладчик.** Если параметру `DbgBreakOnError` присвоено ненулевое значение, инфраструктура переходит в отладчик (если установлен и включен) при каждом вызове драйвером функции `WdfVerifierDbgBreakPoint`. При нулевом значении параметра `DbgBreakOnError` переход в отладчик не происходит. Если параметр `DbgBreakOnError` не существует, но значению параметра реестра `DbgBreakOnError` присвоено ненулевое значение, то инфраструктура переходит в отладчик при каждом вызове драйвером функции `WdfVerifierDbgBreakPoint`.
- ◆ **Отслеживание дескрипторов.** Чтобы включить отслеживание обращений к дескрипторам объектов одного или нескольких типов, установите значение параметра реестра `TrackHandles`. Когда этот параметр включен, KMDF отслеживает ссылки на объекты указанных типов. Эта возможность способствует обнаружению утечек памяти, вызванных неосвобожденными ссылками.

Чтобы включить отслеживание ссылок, в редакторе реестра `RegEdit` присвойте ненулевое значение параметру `TrackHandles` в подразделе `Parameters\Wdf` раздела `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Service` реестра Windows.

Так как параметр `TrackHandles` является мультистроковым (типа `REG_MULTI_SZ`), то можно указать один или несколько типов объектов WDF, как показано в следующем примере:

```
TrackHandles    MULTI_SZ:    WDFDEVICE    WDFQUEUE
```

Эта настройка указывает KMDF отслеживать дескрипторы объектов типов `WDFDEVICE` и `WDFQUEUE`. Отслеживание объектов KMDF всех типов указывается звездочкой (\*).

## Использование информации от KMDF Verifier при отладке

После загрузки драйвера определить, работает ли KMDF Verifier, можно с помощью следующей команды расширения отладчика:

```
!wdfkd.wdfdriverinfo DriverName 0x1
```

Расширение отладчика `!wdfkd.wdfdriverinfo` возвращает информацию о драйвере. В значении после имени драйвера указывается набор флагов, задающих, какую информацию нужно возвратить. Значение `0x1` возвращает состояние верификатора KMDF Verifier. Если KMDF Verifier активирован, то отладчик возвращает следующую информацию:

- ◆ имя образа драйвера;
- ◆ имя образа библиотеки WDF;
- ◆ адрес внутренней переменной `FxDriverGlobals`;

- ◆ значение внутренней переменной `WdfBindInfo`;
- ◆ номер версии KMDF, под которую скомпилирован драйвер.

Номер версии не должен быть большим, чем версия KMDF, установленная на тестовой системе.

Для примера, в листинге 21.3 показан вывод расширения отладчика `!wdfkd.wdfdriverinfo` для драйвера `WdfRawBusEnumTest`.

#### Листинг 21.3. Пример вывода расширения отладчика

```
0: kd> !wdfdriverinfo wdfrawbusenumtest f
-----
Default driver image name:      wdfrawbusenumtest
WDF library image name:        Wdf01000
FxDriverClobals    0x83b6af18
WdfBindInfo        0xf22550ec
Version           v1.5 build(1234)
-----
WDFDRIVER: 0x7cfb30d0
!WDFDEVICE 0x7c58b1c0
    context: dt 0x83a74ff8 ROOT_CONTEXT (size is 0x1 bytes)
    <no associated attribute callbacks>
!WDFDEVICE 0x7d2df1c8
    context: dt 0x82d20ff0 RAW_PDO_CONTEXT (size is 0xc bytes)
    <no associated attribute callbacks>
!WDFDEVICE 0x7c8671d8
    context: dt 0x83798fe0 PDO_DEVICE_DATA (size is 0x1c bytes)
    EvtCleanupCallback f2251710 wdfrawbusenumtest!RawBus_Pdo_Cleanup
-----
WDF Verifier settings for wdfrawbusenumtest.sys is ON
Pool tracking is ON
Handle verification is ON
IO verification is ON
Lock verification is ON
Handle reference tracking is ON for the following types:
    WDFDEVICE
```

Для команды в предыдущем примере было указано значение флага `0xF`, таким образом включающее в вывод состояние верификатора KMDF Verifier и всей прочей связанной информации. Вывод, показанный в примере, также содержит информацию о контексте и функциях обратного вызова, ассоциированных с каждым дескриптором объекта типа `WDFDEVICE`, т. к. установлено ненулевое значение параметра реестра `TrackHandles`.

## Верификатор UMDF Verifier

Инфраструктура UMDF содержит встроенный верификатор, который выявляет проблемы в инфраструктуре и коде драйверов UMDF. Верификатор UMDF Verifier всегда активирован как в свободных, так в проверочных сборках среды, поэтому не нужно ничего делать, чтобы его включить. Так как UMDF Verifier всегда включен, то проблемы в коде драйверов UMDF всегда фатальные. То есть если только к процессу, исполняющему драйвер UMDF, не подключен отладчик, то хост-процесс перестает отвечать.

Верификатор UMDF Verifier проверяет код драйвера UMDF на наличие некорректных вызовов интерфейсов драйвера устройства UMDF. Также он проверяет число закрытых вызовов в самой инфраструктуре.

В табл. 21.5 приводится список ошибок, которые может выявить UMDF Verifier, с примерами методов, которые проверяются на данную ошибку.

**Таблица 21.5. Ошибки, которые может выявить UMDF Verifier**

Ошибка	Пример метода
Передача <code>NULL</code> вместо параметра	Передача <code>NULL</code> вместо параметра методу <code>WDFDevice::CreateSymbolicLink</code> , <code>IWDFDevice::GetDefaultIoTarget</code> или <code>IWDFDevice::GetDriver</code>
Передача пустой строки вместо параметра	Передача в параметре методу <code>IWDFDevice::CreateSymbolicLink</code> указателя на пустую строку
Передача нулевого размера в параметре	Передача методу <code>IWDFMemory::SetBuffer</code> нулевое значение для размера буфера
Передача значения, не определенного в перечислении	Передача в первом параметре методу <code>IWDFDevice::SetPnPState</code> значения, не являющегося членом перечисления <code>WDF_PNP_STATE</code>
Попытка завершить уже завершенный запрос	Вызов метода <code>IWDFIoRequest::CompleteWithInformation</code> или <code>IWDFIoRequest::Complete</code> для уже завершенного запроса
Попытка разрешить или запретить отмену запроса ввода/вывода, который не был доставлен из очереди ввода/вывода	Вызов метода <code>IWDFIoRequest::UnmarkCancelable</code> для запроса ввода/вывода, который не был доставлен из очереди
Попытка получить объект памяти, который представляет буфер ввода для запроса ввода/вывода, не являющегося запросом на чтение или запросом IOCTL	Вызов метода <code>IWDFIoRequest::GetInputMemory</code> для запроса ввода/вывода на чтение
Попытка отформатировать для операции записи объект запроса ввода/вывода на канале USB, не являющийся конечной точкой OUT	Вызов метода <code>IWDFUsbTargetPipe::FormatRequestForWrite</code> для объекта запроса ввода/вывода на канале USB, являющегося конечной точкой IN

## Остановы bugcheck UMDF

Когда UMDF обнаруживает ошибку, он вызывает останов `bugcheck` в хост-процессе. В отличие от останова `bugcheck` режима ядра, останов `bugcheck` UMDF не вызывает фатальный сбой системы. Вместо этого UMDF выполняет такую последовательность действий:

- Создает файл дампа памяти и сохраняет ее в папке для файлов журналов компьютера, например, `%windir%\System32\LogFiles\WUDF\Xxx.dmp`.
- Создает отчет об ошибке, который пользователь может отослать корпорации Microsoft.
- Подключает отладчик (если установлен и включен). Отладчик выводит сообщение об ошибке, подобное показанному в листинге 21.4.
- Останавливает хост-процесс и отключает устройство.

В листинге 21.4 приведен пример вывода отладчика для ошибки, выявленной верификатором UMDF Verifier. В примере драйвер UMDF вызвал метод `IWDFDevice::CreateSymbolicLink` с указателем на нулевую строку, что является недействительным параметром для этого метода.

#### Листинг 21.4. Пример вывода отладчика для UMDF Verifier

```
**** WUDF DriverStop - Driver error 0x501000100000f34
**** in Host
**** z:\umdf\drivers\wdf\umdf\driverhost\framework\
      wudf\wdfdevice.cpp:3892 (CWdfDevice::CreateSymbolicLink):
Invalid input parameter
**** (b)reak repeatedly, (c)ontinue, (d)ump stack, (i)gnore or (t)erminate process?
```

## Извещение об ошибках UMDF

UMDF докладывает об ошибках, обнаруженных верификатором UMDF Verifier, а также о других проблемах посредством механизма Windows Error Reporting (WER). Кроме ошибок, обнаруженных верификатором UMDF Verifier, UMDF также докладывает о необработанных исключениях в хост-процессе, неожиданных завершениях хост-процессов и сбоях или превышениях лимита времени в критических операциях.

Содержимое доклада WER UMDF зависит от конкретной проблемы. В общем, отчет об ошибке может содержать следующую информацию:

- ◆ дамп памяти хост-процесса;
- ◆ копию внутреннего журнала трассировок UMDF;
- ◆ информацию о конфигурации устройства, которая может включать имя устройства и производителя, а также имена и версии установленных для устройства драйверов;
- ◆ внутренний анализ проблемы, который может содержать адрес последнего вызова драйвера к инфраструктуре (или наоборот), код проблемы, информацию об исключении и т. п.

Дополнительную информацию о том, как UMDF и операционная система обрабатывают ошибки, см. в разделе **Handling Driver Failures** (Обработка сбоев драйверов) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79794>. Подробности о механизме WER см. в разделе **Windows Error Reporting: Getting Started** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79792>.

## Верификатор Application Verifier

Инструмент для динамической верификации Application Verifier (AppVerifier.exe) применяется для верификации приложений с неуправляемым кодом. Его можно использовать для обнаружения ошибок в драйверах пользовательского режима и в любых приложениях пользовательского режима, поставляемых вместе с драйверами режима ядра или пользовательского режима. Верификатор Application Verifier является ценным инструментом для улучшения качества всего драйверного пакета.

Он может обнаруживать неочевидные ошибки программирования, которые иначе было бы трудно выявить во время обычного тестирования приложения или драйвера. Верификатор

Application Verifier можно использовать самостоятельно или совместно с отладчиком пользовательского режима на системах под управлением Windows XP или более поздними версиями Windows.

Application Verifier можно загрузить с Web-сайта Microsoft Download Center по адресу <http://go.microsoft.com/fwlink/?LinkId=79601>.

## Как работает Application Verifier

Application Verifier отслеживает действия приложения во время его исполнения, подвергает приложение различным нагрузкам и тестированием и генерирует отчет о возможных ошибках в работе или исполнении приложения.

Application Verifier предназначен специально для обнаружения и отладки искажений в памяти и критических уязвимостей безопасности. Он отслеживает и докладывает о взаимодействии приложения с операционной системой Windows и профилирует использование приложением объектов, реестра, файловой системы и Windows API, включая кучи, дескрипторы и блокировки. Он также позволяет выполнять проверки для предсказания, как хорошо приложение будет работать под управлением User Account Control (UAC) в Windows Vista.

## Использование Application Verifier для верификации драйверов UMDF

Application Verifier устанавливается в папку %windows%\system32.

Для работы с Application Verifier с командной строки, введите команду appverif хотя бы с одним параметром. Если ввести команду appverif без параметров, то запустится приложение Application Verifier, в котором можно выбрать приложения для тестирования и сами тесты, а также просмотреть справочную информацию.

Для верификации драйвера UMDF Application Verifier необходимо включить для хост-процесса (WUDFHost.exe) UMDF, в котором также исполняется и драйвер, который нужно проверить. Проверку можно сфокусировать на драйвере, а не на саму UMDF, с помощью команды, подобной следующей:

```
appverif -enable handles locks heaps memory exceptions TLS -for WUDFHost.exe
```

В данной команде параметры -enable, handles, locks, heaps, memory, exceptions, TLS задают тесты, которые необходимо выполнить. А именно:

- ◆ handles — проверка, что приложение не будет пытаться использовать недействительные дескрипторы;
- ◆ locks — проверка на корректное использование приложением блокировок файлов. Основной целью теста locks является обеспечить, что приложение правильно использует критические разделы;
- ◆ heaps — выполняет проверку на искажения в куче;
- ◆ memory — проверяет на правильное использование приложением функций для манипулирования виртуальным адресным пространством (таких как VirtualAlloc, VirtualFree и MapViewOfFile);
- ◆ exceptions — проверяет, чтобы приложение не скрывало нарушений доступа посредством структурной обработки исключений;

- ◆ TLS — проверяет, чтобы приложение правильно использовало функции для работы с локальной памятью потока.

В предыдущем примере последний параметр `-for WUDFHost.exe` команды `appverif` включает Application Verifier для хост-процесса, в котором исполняется проверяемый драйвер.

### Полезная информация

Команда контроля качества UMDF использует Application Verifier в качестве стандартного инструмента для верификации самой UMDF. Тесты, указанные в примере запуска верификатора, были стандартными установками, с которыми выполнялось тестирование UMDF для Windows Vista.

Для получения всех подробностей о тестах и процедурах для Application Verifier см. интерактивную справку для этого инструмента. Для этого запустите Application Verifier и выберите пункт **Help** в меню **Help**, или же нажмите клавишу **<F1>**.

## Оптимальные методики для тестирования драйверов WDF

### Советы по сборке драйверов

При сборке драйверов примите во внимание следующие оптимальные методики.

- ◆ **Выполняйте сборку и тестирование драйвера для всех версий Windows, для которых он предназначается.** Сборка для самой ранней версии Windows предоставляет двоичный файл драйвера, совместимый также и со всеми дальнейшими версиями Windows. Сборка для самой последней версии Windows предоставляет самые последние проверки для драйвера.
- ◆ **Выполняйте сборку драйвера как для 32-битных, так и для 64-битных платформ.** Сборка для этих двух типов платформ может помочь выявить проблемы в коде драйвера. В особенности, компилирование для 64-битных платформ может помочь выявить проблемы, связанные с указателями, тонкими различиями в компиляторах и инструкциями встроенного ассемблера, которые иногда вставляются по оплошности.

Дополнительную информацию на эту тему см. в разделе **Compiler for x64 64-Bit Environments** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=79791>.

- ◆ **Во время разработки выполняйте сборку драйвера в среде проверочной сборки.** Среда проверочной сборки создает драйвер с разрешенным отладочным кодом. Проверочные сборки облегчают процесс отладки, т. к. компилятор не оптимизирует двоичные файлы так тщательно, как со свободными сборками.
- ◆ **Для тестирования на производительность и для поставки выполняйте тестирование драйверов в среде свободной сборки.** Среда свободной сборки создает рабочий драйвер с оптимизированным рабочим кодом и отключенными отладочными кодом.
- ◆ **Пользуйтесь возможностями компилятора для контроля ошибок.** Выполните компиляцию и сборку драйвера, рассматривая предупреждения как ошибки и включив уровень ошибок `/w4`. Это поможет выявить проблемы в исходном коде в самой полной мере, на которую способен компилятор.

Подробная информация и дополнительные оптимальные методики по этому вопросу приводятся в *главе 19*.

## Советы по использованию инструментов наилучшим образом

При использовании инструментов примите во внимание следующие оптимальные методики.

- ◆ **Обязательно используйте самые новые инструменты.** При отладке драйверов всегда следует пользоваться самыми последними версиями набора разработчика WDK и инструментов Debugging Tools for Windows.

Информация о том, как получить текущие версии наборов и инструментов, предоставлена в *главе 1*.

- ◆ **Выполняйте тестирование драйверов всеми инструментами, которыми можете.** Опустив тестирование одним из возможных инструментов, вы можете пропустить в драйвере серьезную ошибку.

Для тестирования драйверов KMDF, как минимум, включите все опции в Driver Verifier и выполните тестирование инструментами Device Path Exerciser и Plug and Play Driver Test. Также выполните тестирование управления энергопотреблением, в котором выполняются операции перехода в состояние ожидания с последующим возвращением в рабочее состояние и перехода в состояние гибернации с последующим возвращением в рабочее состояние. Для этого теста можно воспользоваться сценарием WDTF Sleep\_Stress\_With\_IO.wsf или инструментом Sleep State Chooser (Sleeper.exe) в папке %wdk%\tools\acpi. Включите KMDF Verifier и изучите информацию, возвращаемую им в отладчике.

Для тестирования драйверов UMDF, как минимум, включите Application Verifier с настройками, описанными ранее в этой главе, и запустите инструмент Device Path Exerciser и сценарии WDTF Disable\_Enable\_With\_IO.wsf и Sleep\_Stress\_With\_IO.wsf

- ◆ **Выполняйте тестирование драйверов как в проверочных, так и в свободных сборках Windows.** Для тестирования с проверочной версией, как минимум, установите проверочную сборку стека вашего драйвера и проверочные версии файлов Ntoskrnl.exe и Hal.dll, как было описано ранее в этой главе.

Выполняйте тестирование драйверов на предмет производительности в свободной сборке.

- ◆ **Используйте расширенные возможности инструментов тестирования.** В особенности выполните тестирование драйвера верификаторами Driver Verifier и Application Verifier с включенными опциями эмуляции низкого уровня ресурсов, чтобы удостовериться, что драйвер работает стабильно и надежно в ситуациях нехватки ресурсов.

- ◆ **Тесты должны быть легко воспроизводимыми.** Тест, выполнение которого требует множества ручных операций или специальной системы, не является практичным. Тест должен быть таким, чтобы его можно было легко выполнять и переносить на дополнительные тестовые машины. Если для выполнения теста требуются ручные операции, необходимо предоставить легко понимаемые письменные инструкции по его выполнению.

- ◆ **Результаты теста должны быть легко подтверждаемыми.** Многие тесты выдают подробные протоколы их исполнения. В этих журналах должна быть одна простая строка, указывающая, было ли тестирование успешным или нет. Журналы, содержащие

большие объемы данных, которые необходимо проверять вручную, не очень полезны для определения, было ли тестирование успешным. По крайней мере, журнал должен быть такого формата, чтобы его можно было легко сравнить с журналом предыдущего успешного тестирования.

## Советы для тестирования драйвера на протяжении его жизненного цикла

Для тестирования драйвера на протяжении его жизненного цикла примите во внимание следующие оптимальные методики.

- ◆ **Выполняйте тестирование на протяжении всего цикла разработки.** Протестируйте разрабатываемый драйвер, как только это возможно, используя такие инструменты, как PRE/fast, SDV и Driver Verifier, и проверочную сборку Windows.
- ◆ **Выполняйте тестирование на разных платформах.** Выполняйте тестирование драйвера на поставляемых версиях Windows как на однопроцессорных, так и на многопроцессорных системах. Если возможно, тестируйте на системах с разными слоями HAL.
- ◆ **Исправив ошибку, создайте регрессионный тест.** При поиске и устраниении причины проблемы создайте регрессионный тест, который можно будет включить в будущее тестирование драйвера. Примите меры к тому, чтобы клиенту никогда больше не пришлось иметь дело с этой ошибкой.
- ◆ **Всегда выполняйте регрессионное тестирование.** Выполните регрессионное тестирование драйверов для каждой дополнительной версии инфраструктуры и для каждого пакета обновлений операционной системы.

### Поддержка поставленных драйверов

Ошибки в поставленных драйверах, возможно, будет трудно отслеживать без достаточных эксплуатационных данных. Сервис WER предоставляет механизм для отправки пользователями отчетов об ошибках в корпорацию Microsoft, которые можно просмотреть на сайте Microsoft Windows Quality Online Services (Winqual). Через сервис WER можно направить пользователей к обновлению Windows, чтобы загрузить новую версию вашего драйвера.

Информация, предоставляемая через сервис WER, охватывает как сбои аппаратной части (т. е. операционной системы), так и сбои программного обеспечения (т. е. приложений), включая информацию о драйверах и приложениях, а также о других модулях, таких как элементы управления и дополнительные модули, которые исполнялись, когда произошел сбой. Данные, предоставляемые сервисом WER, включают небольшой файл дампа сбоя и дополнительную информацию, зависящую от типа ошибки.

Разработчик драйвера предоставляет Winqual информацию для ассоциации отчетов об ошибках его драйвера с его компанией. Если сервис WER имеет доступ к информации о символах, он может исследовать дамп в поисках символа, который вызвал сбой, что позволяет выполнять более точную категоризацию ошибок.

Сервис WER также предоставляет механизм, с помощью которого пользователю можно предложить решение для исправления ошибки. Когда пользователь подает отчет об ошибке, WER может вывести сообщение, чтобы, например, направить пользователя к сайту поддержки поставщика или к сайту обновлений Windows (Windows Update).

Подробности о механизме WER см. в разделе **Windows Error Reporting: Getting Started** на сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79792>.

## ГЛАВА 22

# Отладка драйверов WDF

Отладка является важной частью процесса разработки драйверов. Разрабатываемые программы всегда содержат ошибки, особенно на ранних стадиях разработки. Отладчики могут также выступать в качестве средства обучения, позволяя исследовать драйверы в процессе исполнения, таким образом помогая понять подробности их работы. В этой главе дается введение в отладку драйверов WDF.

В частности, в ней представляются инструменты для отладки драйверов, особенно наиболее широко употребляемый отладчик — WinDbg. В главе также описывается, как подготовить систему к отладке драйверов UMDF и KMDF, и предоставляется подробный пошаговый разбор простого сеанса отладки как драйвера UMDF, так и драйвера KMDF.

Ресурсы, необходимые для данной главы	Расположение
<b>Инструменты и файлы</b>	
Пакет Debugging Tools for Windows	<a href="http://go.microsoft.com/fwlink/?LinkId=80065">http://go.microsoft.com/fwlink/?LinkId=80065</a>
<b>Образцы драйверов</b>	
Fx2_Driver	%wdk%\src\umdf\usb\fx_2driver
Osrusbf2	%wdk%\src\kmdf\osrusbf2
<b>Документация</b>	
Файл справки пакета Debugging Tools for Windows	Поставляется с пакетом Debugging Tools for Windows

## Обзор инструментов отладки для WDF

Со всего разнообразия инструментов для выявления ошибок в драйверах WDF основным является отладчик WinDbg. Этот отладчик оснащен графическим пользовательским интерфейсом и может применяться для отладки как драйверов KMDF, так и драйверов UMDF. Несколько других инструментов применяются для более специализированных целей. В этом разделе рассматриваются инструменты для отладки и их применение для отладки драйверов WDF.

### Примечание

Пакет Debugging Tools for Windows можно загрузить с Web-сайта WHDC. Пакет содержит отладочные инструменты и документацию к ним.

Подробная информация о том, как можно приобрести пакет Debugging Tools for Windows, изложена в *главе 1*.

## Отладчик WinDbg

Предпочитаемым отладчиком как для драйверов UMDF, так и для драйверов KMDF является WinDbg. Этот автономный отладчик применяется для отладки скомпонованных и установленных драйверов. WinDbg оснащен графическим интерфейсом пользователя, который поддерживает стандартные отладочные возможности, такие как установка точек прерывания, просмотр переменных, вывод стека вызовов, пошаговое выполнение исходного кода и т. д. Кроме этого, WinDbg может исполнять из командной строки команды отладчика и команды расширений отладчика, которые предоставляют подробную информацию о многих специфичных для драйвера вопросах.

### Примечание

Хотя применение отладчика Visual Studio с драйверами UMDF может быть технически возможным, этот отладчик не предназначен для отладки драйверов и такое применение не поддерживается корпорацией Microsoft. Кроме этого, набор разработчика WDK содержит коллекцию полезных расширений отладчика UMDF, которые можно использовать только с WinDbg.

С помощью отладчика WinDbg можно выполнять следующие основные виды отладки драйверов WDF.

- ◆ **Отладка в пользовательском режиме.** WinDbg подключается к хост-процессу WUDFHost драйвера UMDF.
- ◆ **Отладка в режиме ядра.** Основной компьютер, с исполняющимся на нем отладчиком WinDbg, подключается к тестовому компьютеру, на котором исполняется драйвер.
- ◆ **Анализы дампов фатальных сбоев.** В случае фатального сбоя драйвера любого типа с помощью WinDbg можно выполнить анализ аварийного дампа.

Хотя WinDbg можно использовать для отладки обоих типов драйверов WDF, для каждого типа применяется свой подход. А именно:

- ◆ отладка драйверов UMDF обычно не такая сложная, как отладка драйверов KMDF. Отладчик и драйвер можно установить на одной и той же системе, а процедуры отладки подобны процедурам для отладки сервиса;
- ◆ для отладки драйверов режима ядра отладчик WinDbg устанавливается на основном компьютере, а драйвер исполняется на тестовом компьютере. Между этими двумя компьютерами необходимо установить какого-либо рода соединение, чтобы отладчик мог общаться с тестовой системой.

## Прочие инструменты

Пакет Debugging Tools for Windows также содержит отладочные инструменты, которые могут служить полезным дополнением к отладчику WinDbg.

- ◆ **Отладчик KD.** Отладчик командной строки KD обладает многими теми же самыми возможностями, что и отладчик WinDbg, но не имеет графического пользовательского интерфейса. Дополнительную информацию об этом отладчике см. в разделе **KD** в файле справки пакета Debugging Tools for Windows.
- ◆ **Инструмент Tlist.exe.** Это приложение командной строки выводит на экран процессы, исполняющиеся на локальном компьютере, вместе с ассоциированной информацией.

WDF содержит два набора расширений отладчика, по одному для UMDF и KMDF. Эти расширения представляют собой небольшие служебные программы, которые исполняются вместе с WinDbg, чтобы предоставить информацию, специфическую для WDF. Дополнительную информацию по этому вопросу см. в разд. "Расширения отладчика" далее в главе.

Отладка обычно выполняется как часть процесса тестирования драйвера. Хотя такие инструменты для тестирования и верификации драйверов, как PREfast и Driver Verifier, не являются частью пакета отладочных средств, их применение типично является частью процесса отладки.

Обзор различных инструментов для тестирования драйверов приводится в главе 21.

## Трассировка WPP

Трассировка WPP обычно применяется для драйверов WDF, чтобы помочь выявить ошибки и их природу. В большинстве образцов драйверов WDF применяется трассировка WPP, таким образом предоставляя многочисленные примеры трассировки.

Подробности трассировки WPP рассматриваются в главе 11.

## Отладка макросов и процедур

Чтобы облегчить выявление ошибок и их природу, в код драйвера можно добавить многочисленные отладочные макросы и процедуры. Некоторые макросы, такие как, например, макрос `ASSERT`, можно использовать с драйверами UMDF и KMDF, в то время как другие можно применять только с драйверами одного типа. Некоторые отладочные возможности являются специфичными для WDF, как рассматривается далее в этой главе.

## Основы отладчика WinDbg

С драйверами UMDF отладчик WinDbg применяется существенно по-другому, чем с драйверами KMDF. Но некоторые базовые возможности и способы работы WinDbg применимы как для драйверов UMDF, так и для драйверов KMDF. Этот аспект отладчика рассматривается далее в этом разделе.

### Примечание

В Windows Vista и более поздних версиях Windows большинство отладочных инструментов должны исполняться с повышенными привилегиями. Самый простой способ для этого — щелкнуть правой кнопкой мыши по значку WinDbg и в контекстном меню выбрать пункт **Run as administrator** (Запуск от имени администратора). Все изложенные в этой главе инструкции по работе с WinDbg полагают, что WinDbg исполняется с повышенными привилегиями в Windows Vista.

## Проверочные и свободные версии сборок

Утилита Build позволяет выполнять два типа сборок: проверочные и свободные. WinDbg можно применять с любым типом сборки, но для отладки предпочтительней работать с проверочными сборками по следующим причинам:

- ◆ большая часть средств оптимизации при компиляции отключена, что облегчает отыскание источников проблем;
- ◆ включен специфический для отладки код, например макрос `ASSERT`.

Основным недостатком проверочных сборок является их пониженная производительность, поэтому они обычно применяются на ранних этапах цикла разработки, когда главной задачей является выявление и устранение ошибок. Но такие задачи, как настройка производительности и окончательное тестирование, должны осуществляться на свободной сборке Windows, и обычно выполняются в конце цикла разработки, после того, как большинство ошибок драйвера было устранено. Хотя WinDbg применяется для отладки любых проблем, возникающих и на последнем этапе разработки, работать со свободными сборками более сложно, и в последних стадиях разработки выявление и исправление ошибок является более трудным.

Тип сборки драйвера и Windows не обязательно должен быть одинаковым. Проверочную сборку драйвера можно исполнять на свободной сборке Windows и наоборот. Потребителям поставляются свободные версии Windows, а проверочные версии предоставляются подписчикам на библиотеке MSDN.

### Полезная информация

Проверочные версии Windows полезно применять на ранних стадиях отладки драйверов. Но наиболее важной практикой для выявления ошибок является всегда включать Driver Verifier.

Подробную информацию относительно того, как получить и пользоваться проверочными сборками, можно найти в разделе **Using Checked Builds of Windows** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79304>.

## Пользовательский интерфейс

WinDbg имеет меню и соответствующую панель инструментов для выполнения таких задач, как организация и исполнение сеанса отладки, пошаговое выполнение исходного кода, установка точек прерывания и т. д. Ключевой частью пользовательского интерфейса является набор окон, которые можно выводить, щелкнув пункт меню **View**. Наиболее важным из этих окон является окно **Command**, которое выводится по умолчанию. В нем отображается информация о текущем сеансе отладки. Также в нем можно вводить команды отладчика и расширения отладчика. На рис. 22.1 показан набор окон пользовательского интерфейса WinDbg для сеанса отладки образца драйвера Fx2\_Driver.

Окно **Command** открывается по умолчанию, и в нем выполняется большая часть работы по отладке. В верхней панели отображаются такие данные, как значения регистров и код ассемблера, генерируемый при пошаговом исполнении кода драйвера. В нижней панели предоставляется поддержка интерфейса командной строки, с помощью которого можно выполнять команды и расширения отладчика. С помощью курсора можно выделить текст в любой панели и скопировать его в командную строку WinDbg. Например, данные, выводимые некоторыми командами расширения отладчика WDF, содержат предлагаемые строки команд

для получения соответствующей информации. Чтобы получить эту информацию, нужно просто скопировать строку команды в нижнюю панель и выполнить команду.

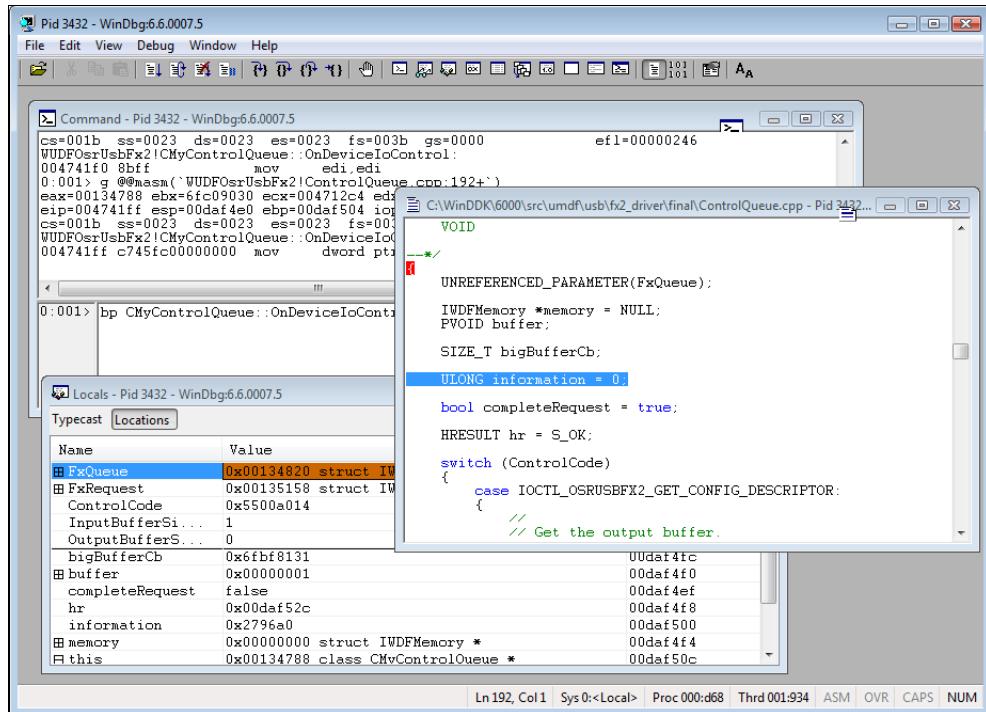


Рис. 22.1. Пользовательский интерфейс отладчика WinDbg

В других окнах выводится подробная информация, и их нужно открывать явным образом через пункт меню **View**. На рис. 22.1 показаны два таких окна: окно **Locals**, в котором отображаются данные локальных переменных процедуры, исполняющейся в настоящее время, и окно исходного кода, в котором можно просматривать и пошагово выполнять исходный код драйвера. Другие окна WinDbg:

- ◆ окно **Watch** — отслеживаются значения указанных переменных;
- ◆ окно **Call Stack** — выводится информация о стеке вызовов;
- ◆ окно **Registers** — отслеживается содержимое регистров.

## Команды отладчика

Некоторые задачи по отладке можно выполнять через графический пользовательский интерфейс. Но через него можно выполнять только сравнительно простые процедуры, такие как, например, пошаговое исполнение исходного кода или выдача инструкции *Break*. А через нижнюю панель окна **Command**, в котором можно выполнять команды и расширения отладчика, можно получить доступ к набору намного более мощных инструментов.

WinDbg имеет набор стандартных команд отладчика. Некоторые команды дублируют функциональность графического пользовательского интерфейса, но многие другие выполняют задачи или позволяют получить подробную информацию о различных аспектах драйвера,

которые невозможно выполнить или получить иным путем. Примеры некоторых распространенных команд отладчика приводятся в табл. 22.1.

**Таблица 22.1. Часто используемые команды отладчика WinDbg**

Тип команды	Команды	Описание
Работа со стековыми фреймами	к и родственные команды	Выводит информацию о стековом фрейме потока и связанную информацию
Для работы с точками прерывания	bp и родственные команды	Используется для установки точек прерываний в исполняющихся драйверах, вывода списка текущих точек прерываний, установки точек прерываний для модулей, которые еще не были загружены, и т. п.
Для работы с памятью	d и родственные команды	Выводит содержимое памяти с учетом таких ограничений, как диапазон, формат вывода, механизм для указания области памяти и т. п.
Для пошагового исполнения	p и родственные команды	Пошагово исполняет код драйвера
Для работы с трассировкой	t и родственные команды	Выполняет трассировку кода драйвера
Переход	g	Прерывает исполнение кода драйвера в отладчике и возвращает драйвер в нормальный режим работы

Полный список команд отладчика см. в разд. "Debugger Commands" в файле справки пакета Debugging Tools for Windows.

## Символы и исходный код

Файлы символов являются необходимыми для эффективной отладки. Они содержат информацию об исполняемом файле, включая имена и адреса функций и переменных. Символы для драйверов UMDF и KMDF хранятся в файле баз данных программы (с расширением PDB). Этот файл создается утилитой Build при компиляции проекта и помещается в папку для выходных файлов проекта.

В случае возникновения трудностей с работой WinDbg, например, если выводится неправильный стек или неправильные значения локальных переменных, вероятной причиной может быть отсутствующие или неправильные файлы символов. Для правильной работы WinDbg требуются правильные файлы символов. В частности, символы должны быть из той самой сборки, в которой создавался драйвер. Например, WinDbg не может использовать символы из свободной сборки для отладки проверочной сборки драйвера. Файлы символов будут разными, даже если они были созданы из абсолютно одинаковых исходных файлов.

### Полезная информация

Оптимизации компилятора, используемые в свободных сборках, могут иногда вызывать вывод в WinDbg неправильных значений локальных переменных даже с правильными символами.

Путь к местонахождению файлов символов драйвера необходимо указывать явным образом, добавив соответствующую папку в список путей поиска символов отладчика WinDbg.

Чтобы задать путь хранения символов драйвера:

1. В WinDbg щелкните пункт меню **File** и выберите команду **Symbol File Path**.
2. В открывшемся диалоговом окне **Symbol Search Path** добавьте в список папку, содержащую символы драйвера. Для разделения папок используется точка с запятой (;).
3. Щелкните кнопку **Reload**, чтобы WinDbg перезагрузил символы, после чего нажмите кнопку **OK**.

### Примечание

Символы можно также перезагрузить, выполнив команду `!d` в окне **Command** отладчика WinDbg.

Также необходимо явно указать путь к файлам символов для Windows. Как и в случае с драйверами, символы Windows должны соответствовать сборке операционной системы.

Символы для Windows можно загрузить с Web-сайта Microsoft и задать путь, как было только что описано. Но получение правильного набора файлов символов иногда может быть сопряжено с трудностями. Предпочтительным способом получения символов для Windows является указать WinDbg путь к открытому серверу символов корпорации Microsoft. Таким образом, WinDbg автоматически загрузит правильные символы для версии Windows, под которой исполняется отлаживаемый драйвер.

### Полезная информация

Обязательно используйте правильные символы для драйвера и Windows. Легче всего получить правильные символы, подключив WinDbg к серверу символов корпорации Microsoft по адресу <http://msdl.microsoft.com/download/symbols>.

**Чтобы указать WinDbg загрузить символы с сервера символов корпорации Microsoft, выполните следующую команду отладчика:**

```
.symfix+
```

Эта команда отладчика добавляет местонахождения сервера символов к пути файла символов и загружает с сервера необходимые PDB-файлы.

Но т. к. объем файлов символов Windows составляет десятки мегабайт, то этот подход практичен только при наличии у вас быстрого Интернета. В противном случае лучше установить символы вручную.

**Чтобы установить символы Windows вручную, скачайте текущие пакеты символов с Web-сайта WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=79331>.**

Для выполнения отладки на уровне исходного кода также необходимо указать WinDbg местонахождение исходных файлов драйвера.

**Чтобы добавить папку с исходными файлами драйвера в путь поиска отладчика WinDbg:**

1. В WinDbg щелкните пункт меню **File** и выберите команду **Source File Path**.
2. В диалоговом окне **Source Search Path** добавьте путь к папке с исходными кодами драйвера в список, после чего нажмите кнопку **OK**. Для разделения нескольких папок используется точка с запятой (;).

## Расширения отладчика

Расширения отладчика работают по большому счету подобно командам отладчика, но предоставляют дополнительную функциональность.

- ◆ WinDbg содержит стандартный набор расширений отладчика в пакете Debugging Tools for Windows.

Например, особенно полезным расширением отладчика является `!analyze`, которое применяется для анализа дампов фатальных сбоев. Некоторые расширения работают как в режиме ядра, так и в пользовательском режиме, в то время как другие работают только в каком-либо одном режиме.

- ◆ WDF содержит набор расширений отладчика, которые предоставляет поддержку для отладки драйверов UMDF и KMDF.

Краткое описание этих расширений приводится в табл. 22.2 и 22.3.

- ◆ Если команды и расширения отладчика, предоставляемые корпорацией Microsoft, не удовлетворяют ваших требований, то можно написать собственные расширения отладчика.

Информацию о том, как создавать собственные расширения отладчика, см. в разд. "Debugger Extension API" в файле помощи для пакета Debugging Tools for Windows.

Расширения отладчика предоставляются в виде DLL-файлов, и WinDbg загружает стандартные расширения автоматически. Другие расширения, включая расширения для WDF, необходимо загружать явным образом. Команды расширений отладчика начинаются с восклицательного знака (!). Они исполняются самостоятельно, как и команда отладчика, вводом их с клавиатуры в нижнюю панель окна **Command**, вместе с необходимыми аргументами.

В табл. 22.2 и 22.3 приводится список некоторых часто используемых расширений отладчика для UMDF и KMDF. Полную справочную информацию по командам и расширениям отладчика и интерфейсу API для создания своих расширений отладчика см. в документации отладчика WinDbg.

**Таблица 22.2. Часто используемые расширения отладчика для UMDF**

Расширение	Описание
<code>!wudfext.help</code>	Выводит все расширения отладчика для UMDF
<code>!wudfext.dumpdevstacks</code>	Выводит стеки устройств в хост-процессе
<code>!wudfext.devstack</code>	Выводит выбранный стек устройств в хост-процессе
<code>!wudfext.dumpirps</code>	Выводит список ожидающих исполнения пакетов IRP в хост-процессе
<code>!wudfext.umirp</code>	Выводит информацию о пакете IRP пользовательского режима
<code>!wudfext.driverinfo</code>	Выводит информацию о драйвере UMDF
<code>!wudfext.wdfdevicequeue</code>	Выводит очереди ввода/вывода для устройства
<code>!wudfext.wdfqueue</code>	Выводит информацию об очереди ввода/вывода
<code>!wudfext.wdfrequest</code>	Выводит информацию о запросе ввода/вывода
<code>!wudfext.wdfobject</code>	Выводит информацию об объекте UMDF и его родительских и дочерних отношениях
<code>!wudfext.pnp</code>	Выводит состояния Plug and Play и управления энергопотреблением

**Таблица 22.3.** Часто используемые расширения отладчика для KMDF

Расширение	Описание
!wdfkd.wdfhelp	Выводит все расширения отладчика для KMDF
!wdfkd.wdfcrashdump	Выводит дамп фатального сбоя, включающий информацию из журнала инфраструктуры
!wdfkd.wdfdevice	Выводит информацию, ассоциированную с дескриптором объекта типа WDFDEVICE
!wdfkd.wdfdevicequeues	Выводит информацию обо всех объектах очереди, принадлежащих указанному устройству
!wdfkd.wdfdriverinfo	Выводит информацию о драйвере WDF, например версию его среды исполнения и иерархию дескрипторов объектов
!wdfkd.wdfhandle	Выводит информацию об указанном дескрипторе KMDF
!wdfkd.wdfiotarget	Выводит информацию об указанном дескрипторе объекта типа WDFIOTARGET
iwdfkd.wdfldr	Выводит список всех загруженных драйверов KMDf
!wdfkd.wdflogdump	Выводит информацию из журнала инфраструктуры
!wdfkd.wdfqueue	Выводит информацию об указанном дескрипторе объекта типа WDFQUEUE
!wdfkd.wdfrequest	Выводит информацию об указанном дескрипторе объекта типа WDFREQUEST

**Примечание**

Несколько команд расширений отладчика для KMDF предоставляет больший объем информации при включенном верификаторе KMDF Verifier, чем без него. Например, команда расширения `!wdfdriverinfo` выводит информацию об утечках дескрипторов. Дополнительную информацию о Driver Verifier см. в главе 21.

Расширения отладчика WDF упакованы в две отдельные библиотеки DLL. Расширения отладчика для UMDF находятся в файле WudfExt.dll, а для KMDF — в файле Wdfkd.dll. Обе библиотеки DLL включены в набор разработчика WDK и расположены в папке %wdk%\bin. Эти библиотеки также включены в отладочный пакет и находятся в папке Program Files\Debugging Tools for Windows\winext. Набор отладчика WDK и пакет Debugging Tools for Windows поставляются раздельно.

**Подготовка к отладке драйверов UMDF**

Драйверы UMDF представляют собой библиотеки DLL пользовательского режима, которые исполняются в экземпляре хост-процесса WUDFHost. Методы для отладки этих драйверов похожи на методы для отладки сервисов. Организовать отладку драйверов этого типа намного проще, чем отладку драйверов режима ядра. В частности, как отладчик WinDbg, так и отлаживаемый драйвер могут быть установлены на одном и том же компьютере. В этом разделе рассматривается, как подготовить систему и начать сеанс отладки.

**Примечание**

Прежде чем начинать отладку, необходимо установить драйвер и отладочные инструменты на компьютер с установленной версией Windows, под которую разрабатывается драйвер.

Дополнительная информация об установке драйверов UMDF приводится в главе 20.

## Как разрешить отладку кода загрузки и запуска драйвера

Процесс отладки начинается с подключения отладчика WinDbg к экземпляру хост-процесса WUDFHost, в котором исполняется драйвер. Это вызывает переход исполнения процесса в отладчик и позволяет выполнять такие задачи, как установка точек прерывания в одном или нескольких методах драйвера. Но код для загрузки и запуска драйвера исполняется сразу же после того, как менеджер драйверов запустит процесс WUDFHost, прежде чем обычно можно подключить отладчик WinDbg к процессу.

Чтобы разрешить отладку кода загрузки и запуска драйвера, WDF предоставляет параметр реестра, который можно установить, чтобы указывать каждому новому экземпляру процесса WUDFHost задерживать исполнение этого кода на достаточный период времени, позволяющий успеть подключить отладчик к процессу WUDFHost. Потом можно будет устанавливать точки прерывания и выполнять отладку кода загрузки и запуска драйвера.

### Чтобы установить период задержки для процесса WUDFHost:

1. Запустите утилиту редактирования реестра RegEdit (Regedit.exe).
2. Перейдите в раздел реестра для процесса WUDFHost (193a1820-d9ac-4997-8c55-be817523f6aa), который находится по следующему пути:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\  
WUDF\Services\{193a1820-d9ac-4997-8c55-be817523f6aa}
```

Изменяя значение параметра `HostProcessDbgBreakOnDriverLoad` этого раздела можно указывать, на сколько секунд процесс WUDFHost должен задержать загрузку драйвера. По умолчанию этому параметру присвоено нулевое значение.

3. Измените значение этого параметра на значение, достаточное, чтобы позволить выполнить подключение отладчика, например, `0xF` (15 секунд).

Параметра `HostProcessDbgBreakOnDriverLoad` достаточно, чтобы решить большинство вопросов отладки загрузки и запуска драйвера. Но задержать исполнение функции `DllMain` с его помощью нельзя. Поэтому для отладки функции `DllMain` или кода инициализации любой глобальной переменной необходимо модифицировать значение параметра `HostProcessDbgBreakOnStart`. Этот параметр указывает процессу WUDFHost задержать исполнение любой части кода драйвера на заданный период времени. Процедура для установки значения параметра `HostProcessDbgBreakOnStart` точно такая же, как и для установки значения параметра `HostProcessDbgBreakOnDriverLoad`.

Кроме этого, можно также установить старший бит значения каждого из этих параметров реестра. В этом случае, если в течение указанного времени инфраструктура успешно не переключит исполнение драйвера в отладчик пользовательского режима, то она попытается подключиться к отладчику режима ядра.

### Полезная информация

Завершив отладку кода загрузки и запуска драйвера, обязательно удалите параметр `HostProcessDbgBreakOnDriverLoad` или `HostProcessDbgBreakOnStart` из реестра. В противном случае загрузка всех драйверов в системе будет медленной.

## Как начать отладку кода загрузки и запуска драйвера UMDF

После установки периода задержки можно начинать отладку.

**Для отладки кода загрузки и запуска драйвера:**

1. Отключите драйвер. Для устройств USB, таких как обучающее устройство OSR USB Fx2, достаточно просто вытащить кабель из разъема USB. Для других устройств отключите драйвер с помощью Диспетчера устройств.
2. Присвойте параметру `HostProcessDbgBreakOnDriverLoad` такое значение, которое даст достаточно времени подключить отладчик WinDbg к процессу WUDFHost.
3. Запустите Диспетчер задач Windows и откройте вкладку **Processes** (Процессы).

На этой вкладке выводятся все экземпляры процесса WUDFHost вместе с их идентификаторами PID (Process Identifier).

### Примечание

Диспетчер процессов не выводит экземпляры процесса WUDFHost и их идентификаторы PID по умолчанию. Чтобы отображать процессы всех пользователей, внизу окна списка процессов необходимо установить флагок **Show processes from all users** (Отображать процессы всех пользователей). А для отображения идентификаторов PID процессов необходимо открыть диалоговое окно **Выбор столбцов** (выполнив последовательно команды **View | Task Manager** (Вид | Выбрать столбцы)) и в нем установить флагок **Идентиф. Процесса (PID)**.

4. Щелкните левой кнопкой мыши по названию столбца **Image Name** (Имя образа), чтобы отсортировать имена процессов таким образом, чтобы все экземпляры процесса WUDFHost отображались вверху списка процесса, для облегчения доступа к ним.
5. Откройте окно командной строки.
6. Введите с клавиатуры следующую частичную команду и нажмите клавишу <Пробел>. Не нажмайте пока клавишу <Enter>!

`WinDbg -p`

7. Начните процесс загрузки драйвера.

Для устройств USB, таких как обучающее устройство OSR USB Fx2, просто подключите кабель USB устройства к разъему USB компьютера. Для других устройств разрешите драйвер с помощью Диспетчера устройств.

8. Наблюдайте в Диспетчере устройств за появлением нового экземпляра процесса WUDFHost в списке процессов.
9. Добавьте идентификатор PID этого нового экземпляра процесса WUDFHost в конец строки команды, начатой в шаге 6, и теперь нажмите клавишу <Enter>. Опция `-p` и идентификатор PID обязательно должны быть разделены пробелом.

### Внимание!

Шаги 6—9 необходимо выполнять до истечения периода задержки, указанного в значении параметра `HostProcessDbgBreakOnDriverLoad`.

Предыдущая процедура подключает отладчик к новому экземпляру процесса WUDFHost, который вызывает отладчик, прежде чем вызовется код для загрузки и запуска драйвера.

Теперь с помощью WinDbg можно установить одну или несколько точек прерывания в коде загрузки и запуска драйвера, после чего выполнить команду `g` для продолжения выполнения процесса WUDFHost. Подробный пример использования этой процедуры приводится в разд. "Пошаговый разбор отладки драйверов UMDF на примере образца драйвера *Fx2\_Driver*" далее в этой главе.

#### Для отладки функции `DllMain` драйвера:

1. Присвойте параметру `HostProcessDbgBreakOnStart` такое значение, которое даст достаточно времени подключить отладчик WinDbg к процессу WUDFHost.
2. Подключите отладчик WinDbg к необходимому процессу WUDFHost, согласно описанной ранее процедуре.
3. После того как процесс WUDFHost перейдет в отладчик, выполните следующую команду, чтобы дать указание процессу WUDFHost перейти в отладчик при загрузке DLL-библиотеки драйвера в память:  
`sxe ld DriverName.dll`
4. Выполните команду `g`, чтобы выйти из отладчика и продолжить исполнение процесса WUDFHost.

После того как процесс WUDFHost загрузит драйвер в память, он перейдет в отладчик, и можно будет устанавливать точки прерываний в функции `DllMain` драйвера или в любом коде глобальной инициализации, для которого необходимо выполнить отладку.

## Отладка исполняющегося драйвера UMDF

Если не требуется выполнять отладку кода загрузки и запуска драйвера, можно просто подключить отладчик WinDbg к соответствующему экземпляру процесса WUDFHost. Но в системе может исполняться несколько экземпляров процесса WUDFHost, поэтому не так просто определить, в каком из этих экземпляров исполняется драйвер, подлежащий отладке. Чтобы найти правильный экземпляр процесса WUDFHost и подключить к нему отладчик WinDbg, воспользуйтесь следующими процедурами.

#### Чтобы определить необходимый экземпляр процесса WUDFHost:

1. В Диспетчере задач откройте вкладку **Processes** (Процессы), чтобы показать экземпляры процесса WUDFHost, исполняющиеся в настоящее время.
2. Откройте окно командной строки.
3. Запустите утилиту `Tlist` с идентификатором PID первого экземпляра процесса WUDFHost. Команда выглядит следующим образом:  
`tlist processID`

В результате на экран будет выведен список всех библиотек DLL, которые были загружены процессом с указанным `processID`.

4. Исследуйте этот список на наличие в нем драйвера, подлежащего отладке.

В случае отсутствия в списке необходимого драйвера повторите шаг 3 для каждого экземпляра процесса WUDFHost из списка Диспетчера процессов, пока требуемый драйвер не будет обнаружен.

5. Запомните идентификатор PID экземпляра процесса WUDFHost, который содержит драйвер, подлежащий отладке.

**Чтобы подключить WinDbg к хост-процессу из командной строки**, запустите отладчик WinDbg из командной строки с идентификатором PID необходимого экземпляра процесса WUDFHost.

#### **Чтобы подключить WinDbg к хост-процессу из самого отладчика:**

1. Запустите отладчик WinDbg, после чего выполните последовательность команд меню **File | Attach to a Process**.
2. Выберите необходимый экземпляр процесса WUDFHost и щелкните кнопку **OK**, чтобы подключить к нему отладчик WinDbg.

## **Отслеживание объектов UMDF и подсчет ссылок**

Распространенной проблемой с программами COM является неправильный подсчет ссылок. В общем, обнаружение ошибок, вызванных неправильным подсчетом ссылок, может быть очень трудной задачей. Но UMDF упрощает процесс подсчета ссылок для драйверов UMDF, поддерживая отслеживание объектов и подсчета ссылок.

Для этого применяются два параметра в разделе реестра для процесса WUDFHost: `TrackObjects` и `TrackRefCounts`. По умолчанию эти параметры имеют нулевые значения, вследствие чего отслеживание отключено. Чтобы включить отслеживание объектов, параметру `TrackObjects` необходимо присвоить значение 1. В случае обнаружения утечки памяти установите значение параметра `TrackRefCounts` в 1, что позволит просмотреть статистику получений и освобождений ссылок.

#### **Полезная информация**

Отслеживание подсчета ссылок значительно ухудшает производительность. Эту возможность следует разрешать только в случае надобности выявить и удалить причину утечки памяти. В противном случае эта возможность должна быть отключена.

Если при выгрузке драйвера счетчик ссылок для какого-либо объекта драйвера имеет ненулевое значение, то инфраструктура переходит в отладчик. В отладчике можно будет использовать расширение отладчика `!umprobjects` для UMDF, чтобы выполнить дамп незавершенных объектов драйвера.

Чтобы использовать эту процедуру, необязательно дожидаться, пока инфраструктура вызовет отладчик. Просматривать незавершенные объекты и счетчики ссылок драйвера с помощью расширения отладчика `!umprobjects` можно в любое время.

Если стандартные параметры раздела реестра для процесса WUDFHost не удовлетворяют всех требований, можно добавить два необязательных параметра.

◆ Параметр `MaxRefCountChangesTracked`. По умолчанию UMDF отслеживает максимум 256 изменений счетчика ссылок для объекта.

Чтобы отслеживать большее число изменений счетчика ссылок, присвойте параметру `MaxRefCountChangesTracked` соответствующее значение.

◆ Параметр `MaxStackDepthTracked`. По умолчанию максимальная глубина отслеживания стека инфраструктурой UMDF равняется 16 фреймам.

Чтобы отслеживать большее число стековых фреймов, присвойте параметру `MaxStackDepthTracked` соответствующее значение.

## Отладка фатального сбоя драйвера UMDF

Проблемный драйвер UMDF не может вызвать фатальный сбой системы, но может это сделать со своим хост-процессом WUDFHost. Такой сбой иногда называется "остановом драйвера" (driver stop). Если к процессу подключен отладчик, то процесс переходит в этот отладчик, и можно будет сразу же выполнить отладку с помощью дампа сбоя. Наиболее полезной для этой цели является команда `!analyze` расширения отладчика. UMDF также создает файл дампа сбоя, который можно исследовать позже. Файл сохраняется в папке `%WINDIR%\System32\Logfiles\Wudf`.

Информацию о том, как выполнять анализ файлов дампа сбоя, см. в разд. *"User-Mode Dump Files"* ("Файлы дампа пользовательского режима") в файле справки пакета Debugging Tools for Windows.

В UMDF также встроен код для выполнения верификации, называющейся UMDF Verifier, который всегда включен. Если драйвер неправильно использует интерфейс DDI UMDF или передает неправильные параметры, UMDF Verifier генерирует останов драйвера. Если подключен отладчик, то можно сразу же выполнить отладку сбоя или же проанализировать файл дампа сбоя позже. Можно также настроить UMDF Verifier для создания отчета об ошибке Windows.

Дополнительную информацию о Driver Verifier см. в главе 21.

## Подготовка к отладке драйверов KMDF

Процесс подготовки для отладки драйверов KMDF более сложный, чем для отладки драйверов UMDF. Для начала, для отладки компонентов режима ядра требуются три составляющие.

- ◆ Основной компьютер, на котором исполняется отладчик WinDbg. Обычно это компьютер, использующийся для разработки и сборки драйвера.
- ◆ Тестовый компьютер под управлением соответствующей сборки Windows, на котором установлен отлаживаемый драйвер и разрешена отладка режима ядра. Отладка обычно выполняется на проверочной сборке драйвера, т. к. выполнять отладку на проверочных сборках намного легче. На тестовом компьютере также часто устанавливают проверочную сборку Windows.
- ◆ Способ для взаимодействия между компьютерами. Традиционно для этой цели последовательные порты основного и тестового компьютеров соединяются нуль-модемным кабелем. В альтернативе можно применить соединение по USB или IEEE 1394.

Прежде чем начинать отладку, необходимо установить драйвер на компьютер с установленной версией Windows, под которую он разрабатывается, и настроить этот компьютер для выполнения отладки. На основной компьютер также необходимо установить отладочные инструменты и соединить оба компьютера соответствующим кабелем.

Далее рассматривается, как подготовить к отладке основной и тестовый компьютеры и как начать сеанс отладки.

### Примечание

С Windows XP и более поздними версиями Windows отладчик режима ядра можно использовать на одной машине с отлаживаемым драйвером. Но этот подход к отладке компонентов

режима ядра предоставляет ограниченные возможности. Так, например, нельзя устанавливать точки прерываний. В этой главе предполагается, что отладка драйвера KMDF выполняется на двух компьютерах.

Подробные инструкции о том, как организовать компьютеры для отладки компонентов режима ядра, см. в разд. "Kernel-Mode Setup" в файле помощи пакета Debugging Tools for Windows.

## Как активировать отладку режима ядра на тестовом компьютере

Отладка режима ядра должна быть явно разрешена в операционной системе Windows тестового компьютера. Среди прочего, это также активирует канал связи между основным и тестовым компьютерами. Процедура для активирования отладки режима ядра зависит от того, работает ли тестовый компьютер под управлением Windows Vista или под более ранней версией Windows.

## Как активировать отладку режима ядра для Windows Vista

В Windows Vista и более поздних версиях Windows конфигурация загрузки находится в хранилище данных BCD (Boot Configuration Data, данные конфигурации загрузки). Данные BCD абстрагируют микропрограммное обеспечение и предоставляют программный интерфейс для манипулирования средой загрузки для всех поддерживаемых Windows аппаратных платформ, включая системы EFI (Extensible Firmware Interface, расширяемый микропрограммный интерфейс).

Легче всего активировать отладку режима ядра, воспользовавшись инструментом BCDEdit, входящим в состав Windows Vista.

### Чтобы активировать отладку режима ядра для системы с Windows Vista:

1. Откройте командное окно, имея повышенные привилегии.
2. Выполните следующую команду:

```
bcdedit /debug on
```

Кроме этого, необходимо выполнить дополнительную команду BCDEdit, чтобы выбрать и настроить канал связи. Например, следующая команда BCDEdit задает канал связи для всех установленных на компьютере операционных систем как соединение IEEE 1394, использующее канал 32:

```
bcdedit /dbgsettings 1394 channel:32
```

Можно также задать настройки для каждой отдельной версии Windows. Подробную информацию на эту тему см. в разделе **BCDEdit Commands for Boot Environment** (Команды BCDEdit для настройки среды загрузки) на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=80914>.

После исполнения этих команд перезагрузите компьютер, чтобы Windows Vista исполнялась в режиме отладки.

### Чтобы отключить отладку ядра, выполните следующую команду:

```
bcdedit /debug off
```

## Как активировать отладку режима ядра для более ранних, чем Windows Vista, версий Windows

Чтобы активировать отладку режима ядра для версий Windows более ранних, чем Windows Vista, необходимо отредактировать системный файл Boot.ini, который находится в корневом каталоге загрузочного диска. В этом файле нужно создать раздел [operating systems], а в нем создать опцию загрузки /debug с одним или несколькими аргументами для настройки канала связи.

В листинге 22.1 приводится пример файла Boot.ini, где в первом элементе раздела [operating systems] указывается загрузка отладочной версии Windows и разрешается канал связи через порт 1394. Во втором элементе задается загрузка Windows XP со стандартной конфигурацией.

**Листинг 22.1. Файл Boot.ini для загрузки отладочной конфигурации Windows**

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=
    "Debugging with 1394" /fastdetect /debug
/debugport=1394 /channel=32
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=
    "Microsoft Windows XP Professional" /fastdetect
```

Для систем, на которых установлено несколько версий Windows в отдельных разделах, в файле Boot.ini создаются добавочные разделы [operating systems], по крайней мере, по одному разделу для каждой версии Windows. В начале процесса загрузки загрузчик Windows выводит на экран все варианты загрузки, из которых можно выбрать необходимый. Если пользователь явно не укажет ни одной конфигурации загрузки, то по умолчанию загрузится первая конфигурация в разделе [operating systems].

Дополнительную информацию по редактированию файла Boot.ini см. в разделе **Boot Options for Driver Testing and Debugging** (Опции загрузки для тестирования и отладки драйверов) в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80622>.

## Подготовка тестового компьютера к отладке драйверов KMDF

Для активирования многих из отладочных возможностей инфраструктуры WDF на тестовом компьютере необходимо установить значения параметров подраздела Parameters\Wdf раздела реестра драйвера. Раздел драйвера имеет то же самое имя, что и драйвер, и находится по следующему пути:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\
    Services\DriverName\Parameters\Wdf
```

В табл. 22.4 приводится краткий обзор параметров подраздела Wdf реестра. По умолчанию все эти параметрыdezактивированы.

### Примечание

Чтобы изменения настроек реестра вступили в силу, драйвер необходимо перезагрузить.

**Таблица 22.4. Параметры реестра для отладки KMDF**

Параметр	Тип параметра	Описание
VerifierOn	REG_DWORD	Чтобы активировать KMDF Verifier, параметру присваивается ненулевое значение
VerifyOn	REG_DWORD	Чтобы активировать макрос WDFVERIFY, параметру присваивается ненулевое значение. Устанавливается автоматически при установке параметра VerifierOn
DbgBreakOnError	REG_DWORD	Чтобы дать указание инфраструктуре вызывать отладчик при вызове драйвером функции WdfVerifierDbgBreakPoint, параметру присваивается ненулевое значение
VerboseOn	REG_DWORD	Чтобы протоколировать подробную информацию в журнале KMDF, параметру присваивается ненулевое значение
LogPages	REG_DWORD	Чтобы указать количество страниц памяти, которые инфраструктура выделяет под журнал, параметру присваивается значение от 1 до 10. Значение по умолчанию — 1
VerifierAllocateFailCount	REG_DWORD	Присваивается ненулевое значение для тестирования ситуаций низкого уровня памяти. Когда параметру присвоено значение <i>n</i> , то после <i>n</i> -го выделения памяти инфраструктура завершает неудачей все последующие попытки выделения памяти драйвером. Этот параметр работает, только если так же установлен параметр VerifierOn
TrackHandles	MULTI_SZ	Содержит имена одного или нескольких типов объектов, для которых требуется отслеживать ссылки. С помощью этого параметра можно выявлять утечки памяти, вызванные неосвобожденными ссылками. Для отслеживания объектов всех типов, параметру присваивается значение *
ForceLogsInMiniDump	REG_DWORD	Присваивается ненулевое значение, чтобы включать журнал KMDF в файл малого дампа памяти при фатальных сбоях системы

## Как начать сеанс отладки KMDF

После активации отладки режима ядра на тестовом компьютере можно начинать сеанс отладки KMDF.

### Чтобы начать сеанс отладки KMDF:

1. Запустите отладчик WinDbg.
2. Переключите WindDbg в режим отладки ядра. Для этого выполните последовательность команд меню **File | Kernel Debug**.

3. В открывшемся диалоговом окне **Kernel Debug** выберите необходимый канал связи с тестовым компьютером и выполните для него необходимые настройки.

Например, чтобы использовать IEEE 1394, следует указать номер канала.

Когда WinDbg работает в режиме отладки ядра, чтобы начать отладку, тестовая система должна перейти в отладчик. Это остановит тестовый компьютер, и управление им будет передано WinDbg.

Далее приводится несколько общепринятых способов заставить тестовую систему перейти в отладчик.

- ◆ **Отладчику WinDbg дается инструкция выполнить переход силой.** Это можно сделать либо выполнив последовательность команд меню **Debug | Break**, либо щелкнув соответствующую кнопку на панели инструментов. Также можно исполнить команду `.break` в окне **Command** отладчика.
- ◆ **С помощью WinDbg динамически вставить точки прерываний в исполняющийся драйвер.** Это довольно гибкий подход, т. к. он позволяет вставлять, активировать, деактивировать или удалять точки прерываний в процессе исполнения сеанса отладки. Эта процедура разбирается в пошаговом рассмотрении отладки UMDF и KMDF далее в этой главе.
- ◆ **Вставить операторы DbgBreakPoint в исходный код драйвера.** Этот подход проще, но менее гибкий, так для того, чтобы вставить точку прерывания, необходимо перекомпилировать и переустановить драйвер.
- ◆ **Драйвер вызывает останов bugcheck, что приводит к фатальному сбою тестового компьютера.** В этот момент можно с помощью WinDbg исследовать данные дампа сбоя. Но для того чтобы восстановить работу компьютера, его необходимо перезагрузить. Вызвать фатальный сбой системы можно, исполнив команду `.crash` в окне **Command**.

После того как тестовая система перейдет в отладчик, можно исследовать переменные, пошагово исполнить исходный код и т. д. Команды отладчика можно вводить только после того, как драйвер перейдет в отладчик. В частности, прежде чем исполнять команду `bp` и родственные команды для динамической установки точек прерывания, может потребоваться принудить переход в отладчик, чтобы можно было исполнить эту команду.

Процесс отладки кода загрузки и запуска драйвера отличается от отладки исполняющегося драйвера, т. к. прежде чем начнется процесс загрузки, необходимо установить, по крайней мере, одну точку прерывания. Один из способов сделать это — вставить жестко закодированную точку прерывания в соответствующую процедуру загрузки или запуска драйвера. Лучшим и более гибким подходом будет установить отсроченную точку прерывания.

#### Для отладки кода загрузки и запуска драйвера KMDF:

1. Запустите отладчик WinDbg.
2. Активируйте отладку ядра, как описано в разд. "Подготовка к отладке драйвера KMDF" ранее в этой главе.
3. С помощью команды `bu` установите отсроченную точку прерывания в соответствующей процедуре, обычно в функции `DriverEntry` или `EvtDriverDeviceAdd`.
4. Начните процесс загрузки драйвера.

Один из способов заставить драйвер загружаться — это отключить, а потом включить драйвер с помощью Диспетчера устройств или инструмента DevCon. Для устройства

USB, такого как устройство обучения Fx2, просто вытащите, а потом вставьте обратно кабель устройства в USB-разъем.

Когда исполнение кода загрузки и запуска драйвера дойдет до точки прерывания, он перейдет в отладчик, после чего можно начинать отладку.

## Как начать отладку фатального сбоя драйвера KMDF

Если ошибка в драйвере вызывает фатальный сбой системы, компьютер необходиомо перезагрузить. Но при этом можно использовать отладчик WinDbg, чтобы попытаться определить причину сбоя, анализируя дамп сбоя одним из следующих способов.

- ◆ Если при сбое тестового компьютера отладчик WinDbg исполняется и подключен, то система переходит в отладчик и можно сразу же начать анализ дампа сбоя.  
Наиболее часто для этой цели используется команда `!analyze` расширения отладчика.
- ◆ Тестовый компьютер можно настроить, чтобы попытаться создать файл дампа сбоя, когда этот сбой произойдет.

В случае успешного создания такого файла, его можно загрузить в отладчик WinDbg и исследовать, пытаясь выяснить причину сбоя. Для этой операции WinDbg не обязательно должен быть подключен к тестовому компьютеру.

Дополнительную информацию на эту тему см. в соответствующих статьях в файле справки пакета Debugging Tools for Windows. В статье "Creating a Kernel-Mode Dump File" ("Создание файла дампа режима ядра") описывается, как создать файл дампа сбоя, а в статье "Using the `!analyze` Extension" ("Использование расширения отладчика `!analyze`") предоставляется информация о том, как использовать расширение отладчика `!analyze` для анализа дампа сбоя.

Одним из полезных инструментов для отладки сбоев драйвера является журнал KMDF. KMDF создает журнал для каждого драйвера, в котором протоколируется история недавних событий, таких как, например, прохождение пакетов IRP по инфраструктуре и соответствующих запросов по драйверу. Дополнительную информацию по этому предмету см. в разд. "Использование WinDbg для просмотра журнала KMDF" далее в этой главе.

## Пошаговый разбор отладки драйверов UMDF на примере образца драйвера Fx2\_Driver

В этом разделе показано, как использовать отладчик WinDbg на примере образца драйвера Fx2\_Driver. В образце нет известных ошибок, но пошаговый разбор исходного кода с помощью WinDbg является удобным способом демонстрации, как использовать этот отладчик с драйверами UMDF.

Это упражнение можно выполнять на том же самом компьютере, на котором драйвер был разработан, но в данном случае полагается, что оно выполняется на отдельном тестовом компьютере. Данный пример был создан со свободной сборкой драйвера Fx2\_Driver, использующегося на компьютере под управлением свободной сборки Windows Vista, но его можно выполнить с таким же успехом под Windows XP.

### Подготовка к сеансу отладки драйвера Fx2\_Driver

Последующие действия основаны на информации, представленной в разд. "Подготовка к отладке драйверов UMDF" ранее в этой главе.

**Чтобы подготовить сеанс отладки:**

1. Выполните сборку драйвера и установите его на тестовом компьютере.
2. Скопируйте исходный код и файлы символов драйвера в удобную папку на тестовом компьютере.
3. Подготовьте тестовый компьютер для выполнения отладки UMDF, как было описано ранее в этой главе.

**Полезная информация**

При применении этого способа в первый раз присвойте параметру `HostProcess-DbgBreakOnStart` большое значение, например 60 (0x3C) или даже 70 (0x78), чтобы дать себе достаточно времени, чтобы подключить WinDbg к процессу WUDFHost.

## Начало сеанса отладки для драйвера Fx2\_Driver

Это пошаговое рассмотрение начинается с кода загрузки и запуска драйвера.

**Чтобы начать сеанс отладки:**

1. Откройте окно командной строки.
2. Введите следующую частичную команду и нажмите клавишу <Пробел>. Не нажимайте пока клавишу <Enter>!

```
WinDbg -p
```

3. В Диспетчере задач Windows откройте вкладку **Processes** (Процессы) и щелкните левой кнопкой мыши по заголовку **Image Name** (Имя образа), чтобы отсортировать список процессов в нисходящем порядке, в результате чего экземпляры процесса WUDFHost должны оказаться вверху списка.
4. Вставьте кабель устройства Fx2 в один из USB-портов компьютера, а в Диспетчере задач Windows наблюдайте за появлением нового экземпляра процесса WUDFHost в списке процессов.

На рис. 22.2 показан пример списка процессов в Диспетчере задач Windows, в котором имеются два экземпляра процесса WUDFHost. Драйвер, рассматриваемый в этом примере, исполняется в верхнем процессе.

5. Добавьте идентификатор PID устройства в конец команды, начатой в шаге 2, после чего нажмите клавишу <Enter>, чтобы запустить отладчик WinDbg и подключить его к процессу WUDFHost.

Этот шаг необходимо выполнить в течение периода задержки, установленного значением параметра `HostProcess-DbgBreakOnStart`.

В примере, показанном на рис. 22.2, идентификатор PID экземпляра процесса WUDFHost для драйвера Fx2\_Driver равен 3432.

## Анализ процедуры обратного вызова *OnDeviceAdd* для драйвера Fx2\_Driver

Подходящим местом для начала отладки кода запуска драйвера будет метод `CMyDriver::OnDeviceAdd` драйвера.

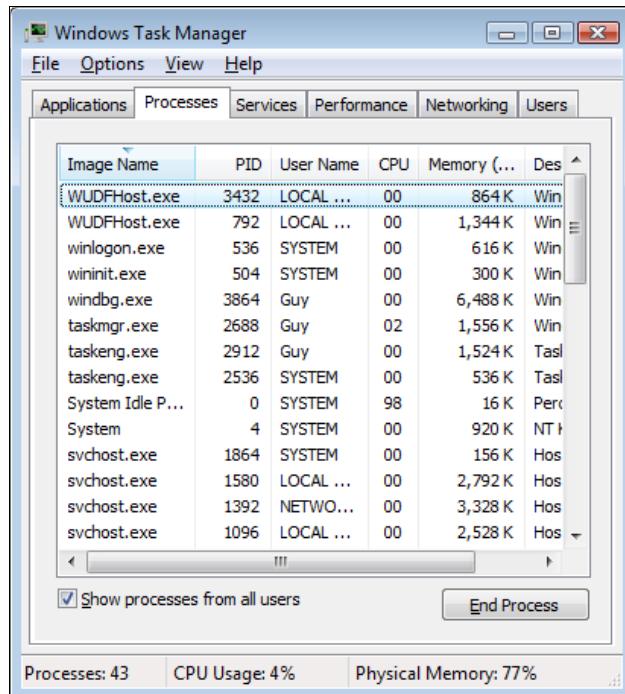


Рис. 22.2. Экземпляры процесса WUDFHost в списке процессов Диспетчера задач Windows

#### Чтобы загрузить символы и исходный код:

1. Задайте пути для папок исходного кода и файла символов драйвера.
2. Исполните команду `lm` отладчика, чтобы удостовериться в том, что имеется правильный драйвер и символы UMDF.
3. В WinDbg щелкните пункт меню **File** и выберите команду **Open Source File**.
4. В открывшемся диалоговом окне **Open Source File** откройте файлы Driver.cpp и Device.cpp. В результате этого появятся два файловых окна.

Файлы Driver.cpp и Device.cpp содержат исходный код для объектов обратного вызова драйвера и устройства соответственно.

#### Чтобы начать отладку кода загрузки драйвера Fx2\_Driver:

1. В окне **Command** отладчика WinDbg введите команду `bu`, чтобы установить отсроченную точку прерывания в начале кода метода `CMYDriver::OnDeviceAdd`:

```
bu CMYDriver::OnDeviceAdd
```

  2. Запустите драйвер, введя в окне **Command** отладчика команду `go`. Когда исполнение драйвера дойдет к этой точке прерывания, WinDbg выскажет в окне исходного кода открывающую скобку метода `CMYDriver::OnDeviceAdd`.
  3. Пошагово исполняйте код, пока не дойдете до следующей строки кода:
- ```
hr = CMYDevice::CreateInstance(FxWdfDriver, FxDeviceInit, &device);
```

Проще всего это сделать с помощью кнопки на панели инструментов. Но можно также использовать команду `p`, которая, кроме исполнения текущей строки, предоставляет некоторые дополнительные возможности.

4. Продолжите пошаговое исполнение кода после этой строки, в которой создается объект устройства.

Объект устройства является одним из ключевых объектов UMDF. С помощью отладчика WinDbg можно получить информацию об этом объекте и его текущем состоянии.

#### **Чтобы получить информацию об объекте устройства и его текущем состоянии:**

1. Откройте окно **Locals** отладчика WinDbg.
2. Разверните узел **device**.

В результате на экране появится содержимое объекта устройства, включая предоставляемые им интерфейсы и значения его элементов данных (рис. 22.3). Обратите внимание на то, что некоторым элементам данных значения еще не были присвоены. Присвоение значений большей части элементов данных происходит позже, в процедуре `CMyDevice::OnPrepareHardware`.

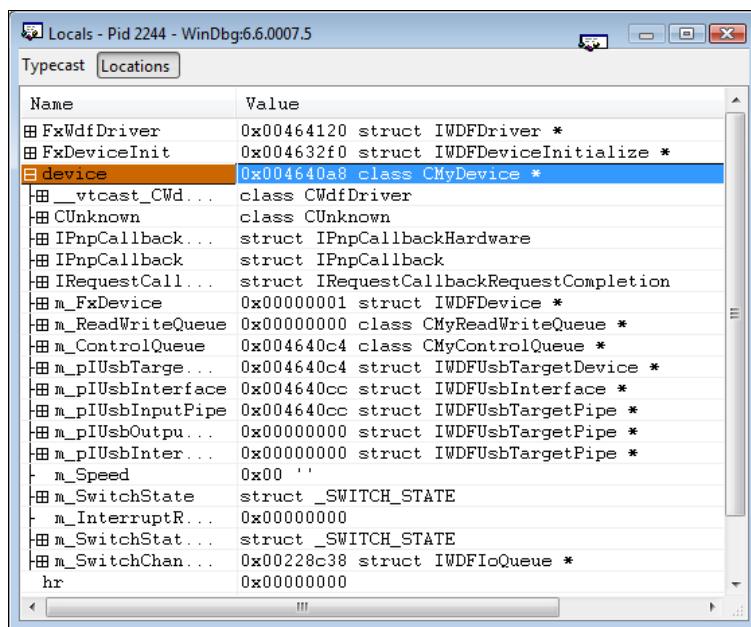


Рис. 22.3. Окно **Locals** с развернутым методом `CMyDevice::CreateInstance`

## **Исследование с помощью расширений отладчика UMDF объекта обратного вызова устройства**

С помощью пользовательского интерфейса отладчика WinDbg можно получить только ограниченный объем информации об объекте устройства. В этом отношении расширения отладчика UMDF часто являются намного более полезными инструментами, т. к. с их помощью можно получить подробную информацию о специфичных для UMDF аспектах.

### Чтобы исследовать объект устройства с помощью расширений отладчика UMDF:

- Если вы этого еще не сделали, то с помощью команды .load загрузите расширения UMDF отладчика, как показано в следующем примере:

```
.load %wdk%\bin\wudfext.dll
```

Вместо `%wdk%` подставьте соответствующий путь к набору разработчика WDK на вашей системе.

- Установите точку прерывания в методе `CMyDevice::OnPrepareHardware` драйвера следующим образом:

```
bp CMyDevice::OnPrepareHardware
```

- Исполните команду `g`, чтобы выполнить драйвер до точки прерывания.

Это остановит отладчик в начале метода `CMyDevice::OnPrepareHardware`. Методы, описанные в этом разделе, можно также использовать для исследования объекта устройства в методе `CMyDevice::OnDeviceAdd`, но метод `CMyDevice::OnPrepareHardware` представляет больший интерес.

Расширение отладчика UMDF `!umpdevstacks` выводит информацию о стеке устройств текущего хост-процесса. Пример исполнения команды `!umpdevstacks` для образца драйвера `Fx2_Driver` показан на рис. 22.4.



Рис. 22.4. Вывод команды расширения отладчика `!umpdevstacks`

## Исследование с помощью расширений отладчика UMDF запроса ввода/вывода

С помощью отладчика WinDbg можно исследовать, как именно драйвер обрабатывает запросы ввода/вывода.

### Чтобы просмотреть обработку драйвером запросов ввода/вывода:

- Установите точку прерывания в методе обратного вызова `CMyControlQueue::OnDeviceIoControl` драйвера.

Этот метод находится в файле `ControlQueue.cpp`. Разные аспекты устройства, включая светодиодную линейку, управляются с помощью запросов IOCTL.

2. Перезапустите драйвер, выполнив команду `g`.
3. Запустите тестовое приложение `Osrusbf2` и с его помощью включите один или несколько светодиодов на светодиодной линейке.

Чтобы доставить запрос драйверу, инфраструктура вызывает метод `CMyControlQueue::OnDeviceIoControl` и переходит в отладчик в начале метода.

4. Чтобы исследовать объект входящего запроса, выполните команду `!wdfrequest` расширения отладчика.

Вывод этой команды для образца драйвера `Fx2_Driver` показан в верхней части рис. 22.5.

```
0:001> !wdfrequest 0x00465e8
CwdfIoRequest 0x00465370
Type: WdfRequestDeviceIoControl
IWDFIoQueue: 0x00464a28
Completed: No
Canceled: No
UM IRP: 0x00246e08 UniqueId: 0x27a Kernel Irp: 0x0x90f148e8
Type: WudfMsg_IOCONTROL
ClientProcessId: 0xffff
Device Stack: 0x001d75a0
IoStatus
    hrStatus: 0x0
    Information: 0x0
Driver/Framework created IRP: No
Data Buffer: 0x00000000 / 0
IsFrom32BitProcess: Yes
CancelFlagSet: No
Cancel callback: 0x00000000
Total number of stack locations: 2
CurrentStackLocation: 2 (StackLocation[ 1 ])
    StackLocation[ 0 ]
        UNINITIALIZED
    > StackLocation[ 1 ]
        IWDFRequest: ???
        IWDFDevice: 0x004642f0
        IWDFFile: 0x0023c0c8
        Completion:
            Callback: 0x00000000
            Context: 0x00000000
        Parameters: (RequestType: WdfRequestDeviceIoControl)
            Input buffer length: 0x1
            Output buffer length: 0x0
            Control code: 0x5500a014
```

**Рис. 22.5.** Вывод команды `!wdfrequest` расширения отладчика для образца драйвера `Fx2_Driver`

Дополнительные идеи о том, как использовать WinDbg для отладки драйверов WDF, см. в разд. "Дополнительные предложения для экспериментирования с WinDbg" в конце этой главы.

## Пошаговый разбор отладки драйверов KMDF на примере образца драйвера Osrusbf2

В этом разделе показано, как использовать отладчик WinDbg на примере образца драйвера KMDF `Osrusbf2`, начиная с исследования процедуры обратного вызова `EvtDriverDeviceAdd`, которая является ключевой частью кода для загрузки и запуска драйвера. Как и в случае с образцом UMDF, в образце драйвера `Osrusbf2` нет известных ошибок, но пошаговый раз-

бор исходного кода с помощью WinDbg является удобным способом демонстрации, как использовать этот отладчик с драйверами KMDF.

## Подготовка к сеансу отладки драйвера Osrusbf2

Последующие действия основаны на информации, представленной в разд. "Подготовка к отладке драйвера KMDF" ранее в этой главе.

### Чтобы подготовить системы к отладке:

1. Выполните сборку драйвера и установите его на тестовом компьютере.
2. Выполните сборку тестового приложения Osrusbf2, после чего скопируйте его в удобную для работы папку на тестовом компьютере.
3. Подготовьте компьютеры для выполнения отладки, как было описано ранее в этой главе. А именно:
  - активируйте на тестовом компьютере отладку режима ядра для версии Windows, исполняющейся на тестовом компьютере;
  - активируйте возможности отладки KMDF в реестре тестового компьютера.

## Начало сеанса отладки для драйвера Osrusbf2

Подготовив системы, можно начинать отладку, следуя процедурам, описанным в разд. "Как начать сеанс отладки KMDF" ранее в этой главе.

### Чтобы начать сеанс отладки:

1. Запустите отладчик WinDbg и переведите его в режим отладки ядра.
2. Выполните команду `break`, чтобы заставить тестовую систему перейти в отладчик.
3. С помощью команды `bp` установите отсроченную точку прерывания в начале процедуры обратного вызова драйвера следующим образом:  
`bp OsrFxEvtDeviceAdd`
4. В отладчике WinDbg откройте окно исходного кода для файла Device.c.
5. Подключите устройство Fx2 к тестовому компьютеру.
6. Выйдите из отладчика на тестовом компьютере, выполнив команду `g`.

Тестовый компьютер начнет загрузку драйвера и перейдет в отладчик, когда инфраструктура вызовет метод `OsrFxEvtDeviceAdd`. При этом открывающая скобка процедуры будет подсвечена в окне с исходным кодом файла Device.c.

## Анализ процедуры обратного вызова `EvtDriverDeviceAdd`

Теперь можно приступить к анализу кода драйвера. Далее приводится несколько примеров, как это делать.

- ◆ Откройте окно **Locals**, выполнив последовательность команд меню **View | Local** отладчика WinDbg.

В этом окне выводятся значения всех локальных переменных.

- ◆ Пошагово исполните код за строкой кода, в которой вызывается метод `WdfDeviceInitSetPnpPowerEventCallbacks`.

- ◆ В окне **Locals** разверните узел **pnpPowerCallbacks**.

Вы должны увидеть, что три обратных вызова имеют ненулевые значения, что означает, что они зарегистрированы в KMDF и будут вызваны в соответствующее время.

- ◆ Пошагово исполните код до строки кода, в которой вызывается метод `WdfDeviceCreate`.

Теперь у нас имеется действительный объект устройства, один из ключевых объектов KMDF.

- ◆ В окне **Locals** разверните узел устройства.

Вы увидите, что в этом узле нет никакой информации. Таким образом, можно сделать вывод, что с помощью окна **Locals** можно получить только ограниченную информацию об объектах KMDF.

Но оно так и предоставляет некоторые очень полезные данные об объекте устройства, а именно — дескриптор объекта. Это шестнадцатеричное значение в столбце **Value** переменной. Само по себе это значение не представляет ничего интересного. Но его можно использовать с расширениями отладчика KMDF, чтобы получить намного больше информации об объекте, чем можно получить через пользовательский интерфейс отладчика.

## Исследование объекта устройства с помощью расширений отладчика KMDF

Расширения отладчика KMDF, описанные в разд. "Расширения отладчика" ранее в этой главе, можно использовать для исследования объекта устройства.

### Чтобы исследовать объект устройства с помощью расширений отладчика KMDF:

1. Если вы еще этого не сделали, то с помощью команды `.load` загрузите расширения KMDF отладчика, как показано в следующем примере:

```
.load %wdk%\bin\x86\Wdfkd.dll
```

2. Чтобы получить информацию об объекте устройства WDF, необходимо получить дескриптор объекта устройства из окна **Locals** и выполнить команду `!wdfdevice` следующим образом:

```
!wdfdevice ObjectHandle [Flags]
```

Аргумент `Flags` определяет выводимую информацию. Результаты исполнения команды `!wdfdevice` расширения отладчика для объекта устройства Fx2 и с аргументом `Flags` равным `0x1F` показаны на рис. 22.6. Это значение аргумента `Flags` устанавливает все флаговые биты и выводит наиболее подробную информацию.

Для получения дополнительной информации о значениях аргумента `Flags`, см. страницы справки для команды расширения отладчика `!wdfdevice` в файле справки пакета Debugging Tools for Windows.

Команда `!wdfdevice` выводит большой объем информации об объекте устройства, включая информацию о его состоянии Plug and Play и состоянии энергопотребления, а также несколько свойств объекта, таких как, например, область синхронизации. Для примера, в листинге 22.6 показан вывод расширения отладчика `!wdfdevice` для образца драйвера Osrusbf2.

```

Command - Kernel 'com:port=com1,baud=57600' - WinDbg:6.6.0007.5
kd> !wdfkd.wdfdevice 0x7d5c71e0 0x1f
Dumping WDFDEVICE 0x7d5c71e0
=====
WDM PDEVICE_OBJECTs:self 82855958, attached 82a97578, pdo 82a97578

Pnp state: 105 ( WdfDevStatePnpInit )
Power state: 300 ( WdfDevStatePowerObjectCreated )
Power Pol state: 500 ( WdfDevStateFwrPolObjectCreated )

Default WDFIOTARGET: 7d567438

No pending pnp or power irps
Device is the power policy owner for the stack

Pnp state history:
[0] WdfDevStatePnpObjectCreated (0x100)
[1] WdfDevStatePnpInit (0x105)

Power state history:
[0] WdfDevStatePowerObjectCreated (0x300)

Power policy state history:
[0] WdfDevStateFwrPolObjectCreated (0x500)

EvtDeviceD0Entry: osrusbf2!OsrFxEvtDeviceD0Entry (f795a4fe)
EvtDevicePrepareHardware: osrusbf2!OsrFxEvtDevicePrepareHardware (f795a2e8)

WDFCHILDLIST Handles:
!WDFCHILDLIST 0x7d7a8340 (static PDO list)

Properties:
SynchronizationScope: WdfSynchronizationScopeNone
ExecutionLevel: WdfExecutionLevelDispatch
IoType: WdfDeviceIoBuffered
FileObjectClass: WdfFileObjectNotRequired
Exclusive: No
AutoForwardCleanupClose: No
DefaultIoPriorityBoot: 0

kd> !wdfkd.wdfdevice 0x7d5c71e0 0x1f

```

Рис. 22.6. Вывод команды !wdfdevice расширения отладчика для образца драйвера Osrusbf2

## Исследование запроса ввода/вывода с помощью расширений отладчика UMDF запроса ввода/вывода

В этом разделе приводится пример использования расширений отладчика KMDF для исследования, каким образом образец драйвера Osrusbf2 обрабатывает запросы ввода/вывода.

### Чтобы просмотреть обработку драйвером запросов ввода/вывода:

1. Установите точку прерывания в функции обратного вызова для события EvtIoDeviceControl драйвера.

Для образца драйвера Osrusbf2 эта функция называется OsrFxEvtIoDeviceControl и находится в файле Ioctl.c. Для управления разными аспектами устройства, включая светодиодную линейку, применяются запросы IOCTL.

2. Перезапустите целевой компьютер, выполнив команду g.
3. Запустите тестовое приложение KMDF на тестовом компьютере и с его помощью включите один или несколько светодиодов на светодиодной линейке.

Чтобы доставить запрос IOCTL драйверу, инфраструктура вызывает функцию OsrFxEvtIoDeviceControl и переходит в отладчик в начале исполнения этой функции.

4. Чтобы исследовать объект входящего запроса, выполните команду !wdfreuest расширения отладчика для дескриптора объекта запроса.

Вывод этой команды для образца драйвера Osrusbf2 показан в верхней части рис. 22.7.

```

kd> !wdfreuest 0x7d7500b0
!IRP 0x82a1e008
!WDFQUEUE 0x7d7a7510
State: Pending. Allocated by WDF for incoming IRP

kd> !IRP 0x82a1e008
Irp is active with 3 stacks 3 is current (= 0x82a1e0c0)
No Mdl: System buffer=82863a58: Thread 82a4ca28: Irp stack trace.
    cmd flg cl Device   File   Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

        Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

        Args: 00000000 00000000 00000000 00000000
>[ e, 0] 0 1 82855958 82a7a2d0 00000000-00000000 pending
    \Driver\osrusbf2
        Args: 00000000 00000001 5500a014 00000000

kd> !WDFQUEUE 0x7d7a7510
Dumping WDFQUEUE 0x7d7a7510
=====
Parallel, Power-managed, PowerOn, Can accept, Can dispatch, Dispatching, Execution
Number of driver owned requests: 1
    !WDFREQUEST 0x7d7500b0 !IRP 0x82a1e008
Number of waiting requests: 0

EvtIoDeviceControl: (0xf795b20e) osrusbf2!OsrFxEvtIoDeviceControl

kd> !WDFQUEUE 0x7d7a7510

```

**Рис. 22.7.** Вывод команды !wdfreuest расширения отладчика для образца драйвера Osrusbf2

Команда расширения отладчика !wdfreuest выводит ограниченный объем информации, но этот вывод содержит полные строки команд для расширений отладчика !IRP и !WDFQUEUE:

- ◆ стандартное расширение отладчика !IRP выводит информацию о пакете IRP, лежащего в основе объекта типа WDFREQUEST;
- ◆ расширение отладчика WDF !WDFQUEUE выводит информацию об указанной очереди ввода/вывода.

Многие расширения отладчика KMDF выводят строки предлагаемых команд, которые упрощают процесс вывода связанной информации. Например, не имея строки для команды !IRP, прежде чем можно было бы выполнить эту команду, было бы нужно определить адрес пакета IRP, что заняло бы некоторое время. Самым простым способом исполнения этих команд будет скопировать строку из верхней панели в нижнюю панель, после чего выполнить эту команду.

В нижней части верхней панели, показанной на рис. 22.7, представлен вывод двух предложенных командных строк.

## Просмотр сообщения трассировки с помощью WinDbg

Отладчик WinDbg можно настроить для получения и вывода сообщений трассировки от драйверов WDF. Таким образом, при отладке драйвера можно просматривать сообщения в окне **Command** отладчика WinDbg в режиме реального времени. В этом примере показано, как перенаправлять сообщения трассировки образца драйвера Osrusbf2 отладчика ядра WinDbg.

Тема трассировки WPP, утилиты TraceView и родственных инструментов и процедур рассматривается в главе 11.

### Чтобы настроить WinDbg для вывода сообщений трассировки:

- С помощью команды .load загрузите файлы Wmitrace.dll и Traceprtd.dll.  
Эти файлы содержат расширения отладчика для трассировки. Находятся файлы в папке Program Files\Debugging Tools for Windows\Winxp.
- Скопируйте TMF-файлы драйвера в удобную папку на основном компьютере.
- С помощью команды расширения отладчика !wmitrace.searchpath укажите местонахождение TMF-файла драйвера.

#### Примечание

Отладчику необходимы TMF-файлы для того, чтобы он мог форматировать сообщения трассировки.

### Чтобы начать сеанс отладки с выводом сообщений трассировки:

- Запустите отладчик WinDbg на основном компьютере и переведите его в режим отладки компонентов ядра.
- Запустите утилиту TraceView на тестовом компьютере и создайте новый сеанс протоколирования трассировки.
- На странице **Log Session Options** утилиты TraceView выберите опцию **Real Time Display**, после чего выберите **Advanced Log Session Options**.
- На вкладке **Log Session Parameter Options** страницы **Advanced Log Session Options** измените значение опции **WinDbg** на **TRUE**, после чего нажмите кнопку **OK**.

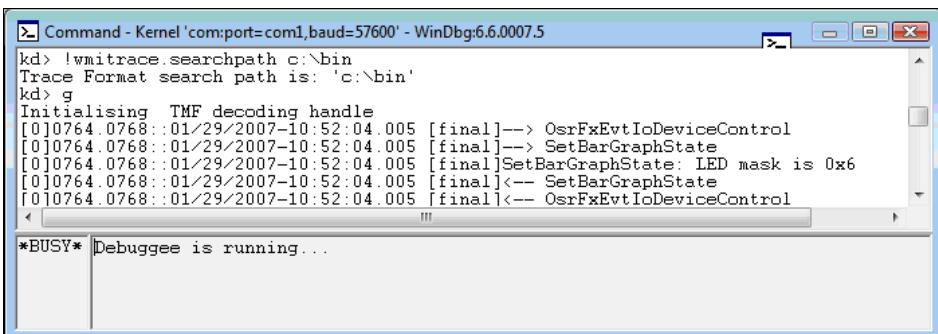


Рис. 22.8. Просмотр сообщений трассировки с помощью WinDbg

5. Завершите процесс создания сеанса протоколирования трассировки, нажав кнопку **Finish**.
6. Запустите тестовое приложение Osrusbf2 и введите несколько изменений в состояние светодиодной линейки, чтобы сгенерировать сообщения трассировки.

На рис. 22.8 показан вывод отладчиком WinDbg сообщений трассировки, сгенерированных включением светодиодов на светодиодной линейке устройства обучения OSR.

## Просмотр журнала KMDF с помощью WinDbg

KMDF имеет внутренний регистратор трассировки, который генерирует журнал для каждого драйвера KMDF. Этот журнал содержит историю недавних событий, таких как, например, прохождение пакетов IRP по инфраструктуре и соответствующих запросов по драйверу. Журнал KMDF можно просматривать и сохранять во время интерактивной отладки с помощью расширений отладчика WDF. Журнал KMDF можно также включить в малый дамп памяти, чтобы можно было исследовать его содержимое после сбоя.

### Для просмотра журнала KMDF во время сеанса трассировки:

1. Если вы еще не сделали это, то загрузите расширения отладчика KMDF, как было описано ранее в этой главе.
2. Установите путь поиска для TMF-файла KMDF.

Файл называется `WdfVersionNumfrer.tmf` и находится в папке `%wdk%\Номер_версии_WDK\tools\tracing\Архитектура`. Чтобы установить путь поиска, выполните команду расширения отладчика `!wdftmffile` с путем к папке с TMF-файлом. В следующем примере устанавливается путь поиска TMF-файла для WDF версии 1.5 из сборки 6000 набора WDK для компьютера под управление 32-битной версии Windows:

```
!wdftmffile %wdk%\6000\tools\tracing\i386\wdf01005.tmf
```

Путь поиска можно также задать, установив переменную среды `TRACE_FORMAT_SEARCH_PATH`. Путь, установленный командой `!wdftmffile`, превосходит по важности путь поиска, устанавливаемый переменной среды.

3. Выведите содержимое файла журнала в окне **Command**, выполнив команду расширения отладчика `!wdfloddump` с именем отлаживаемого драйвера, но без расширения `SYS`.

Например, чтобы сбросить содержимое журнала KMDF для образца драйвера Osrusbf2, выполните следующую команду:

```
!wdfloddump osrusbf2
```

Образец вывода журнала KMDF для драйвера Osrusbf2 показан на рис. 22.9.

Содержимое журнала KMDF можно также сохранить в виде файла журнала трассировки, выполнив команду `!wdflogsav` следующим образом:

```
!wdflogsav [имя_драйвера [имя_файла]]
```

Вместо параметра `имя_драйвера` подставьте имя отлаживаемого драйвера, а вместо параметра `имя_файла` — имя сохраняемого файла журнала. Если параметр `имя_файла` не указан, то файлу присваивается имя по умолчанию `имя_драйвера.etl`.

```

Command - Kernel 'com:port=com1,baud=57600' - WinDbg:6.6.0007.5
Trace format prefix is: %?!u!: %!FUNC! -
TMF file used for formatting IFR log is: c:\winddk\6000\tools\tracing\i386\wdf01005
Log at 827d0000
Gather log. Please wait, this may take a moment (reading 4032 bytes).
% read so far ... 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
There are 110 log entries
--- start of log ---
180: FxIoQueue::DispatchEvents - Driver has configured WDFQUEUE 0x7D5551E8 for Wdf
181: FxFxPkgGeneral::Dispatch - WDFDEVICE 0x7D61E930 !devobj 0x82A81F00 0x00000002(IF
182: FxFxPkgGeneral::Dispatch - WDFDEVICE 0x7D61E930 !devobj 0x82A81F00 0x00000000(IF
183: FxFxPkgIo::Dispatch - WDFDEVICE 0x7D61E930 !devobj 0x82A81F00 0x0000000e(IPR_MJ
184: FxDevice::AllocateRequestMemory - Allocating FxRequest* 82AF9A68, WDFREQUEST*
185: FxIoQueue::QueueRequest - Queuing WDFREQUEST 0x7D506590 on WDFQUEUE 0x7D622A28
186: FxPowerIdleMachine::ProcessEventLocked - WDFDEVICE 0x7D61E930 !devobj 0x82A81F00
187: FxPowerIdleMachine::TimedOutIoIncrement - WDFDEVICE 7D61E930 idle (in D0) not
188: FxPowerIdleMachine::ProcessEventLocked - WDFDEVICE 0x7D61E930 !devobj 0x82A81F00
189: FxIoQueue::DispatchEvents - Thread 82A6BA50 is processing WDFQUEUE 0x7D622A28
190: FxIoQueue::DispatchRequestToDriver - Calling driver EvtIoDeviceControl for WDE
191: imp_WdfRequestRetrieveInputBuffer - Enter: WDFREQUEST 0x7D506590
192: imp_WdfUsbTargetDeviceSendControlTransferSynchronously - WDFUSBDEVICE 7D57C108
193: imp_WdfUsbTargetDeviceSendControlTransferSynchronously - WDFUSBDEVICE 7D57C108
194: FxIoTarget::SubmitSync - WDFIOTARGET 7D57C108, WDFREQUEST F3023990
195: FxIoTarget::SubmitSync - action 0x1
196: FxIoTarget::SubmitSync - Sending WDFREQUEST F3023990, Irp 827EC3A0
197: FxIoTarget::RequestCompletionRoutine - WDFREQUEST F3023990
    ...
kd>

```

Рис. 22.9. Содержимое журнала KMDF для образца драйвера Osrusbfx2

## Получение информации протоколирования после останова bugcheck

С помощью команды `!wdfcrashdump` можно иногда получить информацию из журнала KMDF после системного останова `bugcheck`. Эту информацию можно получить только в том случае, если KMDF определит, что останов `bugcheck` был вызван отлаживаемым драйвером или если в реестре был установлен параметр драйвера `ForceLogsInMiniDump`. Если при останове `bugcheck` к системе подключен отладчик, то с помощью команды `!wdfcrashdump` можно сразу же просмотреть содержимое журнала KMDF. В противном случае эту информацию можно просмотреть, загрузив файл дампа памяти.

KMDF может определить порождение определенным драйвером остановов `bugcheck` с кодами, перечисленными в табл. 22.5.

Таблица 22.5. Коды остановов `bugcheck`

| Код                                 | Значение |
|-------------------------------------|----------|
| DRIVER_IQOL_NOT_LESS_OR_EQUAL       | 0xD1     |
| IQOL_NOT_LESS_OR_EQUAL              | 0xA      |
| KERNEL_APCT_PENDING_DURING_EXIT     | 0x20     |
| KERNEL_MODE_EXCEPTION_NOT_HANDLED   | 0x8E     |
| KMODE_EXCEPTION_NOT_HANDLED         | 0x1E     |
| PAGE_FAULT_IN_NONPAGED_AREA         | 0x50     |
| SYSTEM_THREAD_EXCEPTION_NOT_HANDLED | 0x7E     |

## Управление содержимым журнала KMDF

Для журнала KMDF можно задать несколько из его аспектов:

- ◆ размер журнала;
- ◆ объем информации, записываемой в журнал;
- ◆ строку-префикс, которая добавляется в начале сообщений, записываемых в журнал.

**Размер журнала.** Как обсуждалось в разд. *"Подготовка тестового компьютера к отладке драйверов KMDF" ранее в этой главе*, количество страниц памяти, которые инфраструктура выделяет для ведения журнала, можно указать, установив соответствующее значение параметра LogPages подраздела Parameters\Wdf раздела реестра драйвера. Таким образом, можно указать значение от 1 до 10 страниц. Не забывайте, что размер файла дампа ограничен. Если размер журнала слишком большой, система может не записать его содержимое в файл дампа сбоя.

**Объем информации.** Объем информации, записываемой в файл журнала KMDF, можно регулировать установкой соответствующего значения параметра VerboseOn подраздела Parameters\Wdf раздела реестра драйвера. При ненулевом значении этого параметра инфраструктура записывает в журнал подробную информацию. Но этой возможностью следует пользоваться только при разработке и отладке драйвера, т. к. активированная она может оказать отрицательное влияние на производительность.

**Префикс к сообщениям.** Каждая строчка в журнале KMDF начинается со строки, которая называется префиксом сообщения трассировки. Регистратор трассировки добавляет эту строку в начало каждого сообщения, записываемого в журнал. По умолчанию префикс содержит стандартный набор элементов данных, но эти значения по умолчанию можно изменить для удовлетворения конкретных требований.

Строку-префикс для драйвера KMDF можно изменить, установив переменную среды TRACE\_FORMAT\_PREFIX или воспользовавшись командой расширения отладчика !wdfsettraceprefix. Установка переменной среды TRACE\_FORMAT\_PREFIX позволяет управлять форматом стандартной информации, регистрируемой механизмом ETW. Содержимое префикса задается строкой форматирования, подобной строке форматирования оператора printf.

Для получения подробностей о том, как создавать строку форматирования, см. раздел **Trace Message Prefix** (Префикс для сообщений трассировки) в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80623>.

Переменную среды можно установить командой, подобной следующей:

```
SetTRACE_FORMAT_PREFIX=%2!s!: %!FUNC!: %8!04x!.%3!04x!: %4!s!:
```

Эта команда создает префикс для сообщений трассировки следующего содержания:

```
SourceFile-LineNumber: FunctionName: ProcessID.ThreadID: SystemTime
```

Чтобы задать строку префикса во время отладки, команда wdfsettraceprefix применяется так:

```
!wdfkd.wdfsettraceprefix String
```

Следующий пример этой команды устанавливает такую же строку-префикс, как переменная среды в предыдущем примере:

```
/wdfkd.wdfsettraceprefix %2!s!: %!FUNC!: %8!04x!.%3!04x!: %4!s!:
```

## Дополнительные предложения для экспериментирования с WinDbg

Представленные примеры с пошаговым разбором показывают только базовые возможности отладчика WinDbg. Кроме этих примеров, попробуйте выполнить следующие упражнения.

- ◆ Пошагово исполните больше кода.

Исследуйте значения переменных и выясните, что делают различные процедуры UMDF или KMDF.

- ◆ Установите точку прерывания в функции обратного вызова драйвера для события запроса на чтение, запись или IOCTL и с помощью тестового приложения считайте или запишите значение или отошлите запрос IOCTL.

Эти функции обратного вызова манипулируют объектами памяти WDF, которые содержат ассоциированные буферы данных.

- ◆ Выполните другие расширения отладчика WDF, чтобы увидеть, какую информацию они возвращают. В особенности испытайте команду `!driverinfo`, которая выводит полезную информацию о драйвере.

Обязательно испытайте любые командные строки, предлагаемые в выводе команд расширений отладчика.

Справочную информацию о расширениях отладчика см. в разд. "Specialized Extensions" в файле помощи для пакета Debugging Tools for Windows.

- ◆ Изучайте документацию по отладке.

Аспекты отладки драйверов слишком многочисленны, чтобы всех их можно было рассмотреть в этой главе.

- ◆ Регулярно посещайте группы новостей, посвященные драйверам, или подпишитесь на какую-либо рассылку по этому вопросу.

Проблемы отладки являются распространенным предметом обсуждения, и вы сможете увидеть, как эти проблемы решаются экспертами в данной области.

## ГЛАВА 23

# Инструмент PREfast for Drivers

Инструмент статического анализа исходного кода PREfast for Drivers позволяет выявлять определенные типы ошибок, которые трудно обнаружить компилятором или обычным тестированием. PREfast является важным инструментом для повышения качества как драйверов WDF, так и драйверов WDM.

В этой главе дается обзор инструмента PREfast, включая подробные объяснения, как с ним работать и как анализировать результаты его работы. В ней также предоставляется информация об аннотациях<sup>1</sup> в исходном коде, которые способствуют более эффективному анализу исходного кода инструментом PREfast. Примеры, показанные в этой главе, были адаптированы из драйверов WDM, но большинство правил и аннотаций инструмента PREfast также применимы к драйверам WDF.

| Ресурсы, необходимые для данной главы                                     | Расположение                                                                                            |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <b>Инструменты и файлы</b>                                                |                                                                                                         |
| PREfast.exe                                                               | %wdk%\tools\pfd                                                                                         |
| SpecStrings.h                                                             | %wdk%\inclapi                                                                                           |
| Driverspecs.h                                                             | %wdk%\inclddk                                                                                           |
| <b>Образцы драйверов</b>                                                  |                                                                                                         |
| Примеры, которые активируют разные предупреждения, генерируемые PREfast   | %wdk%\tools\pfd\samples                                                                                 |
| Исходный код драйвера, демонстрирующий специфические для драйвера правила | %wdk%\tools\pfd\samples\fail_drivers                                                                    |
| <b>Документация WDK</b>                                                   |                                                                                                         |
| Документация для PREfast for Drivers                                      | <a href="http://go.microsoft.com/fwlink/?LinkId=80079">http://go.microsoft.com/fwlink/?LinkId=80079</a> |

## Введение в PREfast

Инструмент для статической верификации PREfast for Drivers выявляет основные ошибки в коде программ на языках C и C++ и специфические ошибки в коде драйверов. Инструмент

<sup>1</sup> В данном случае под аннотациями имеются в виду макросы, которые вставляются в исходный код. — Пер.

PREfast for Drivers поставляется как автономный инструмент в наборе разработчика драйверов WDK.

Инструмент PREfast может быть очень полезным при разработке драйверов, т. к. он может найти ошибки, трудно поддающиеся тестированию и отладке, а также выявить предположения программиста, которые могут не всегда быть правильными. Инструмент PREfast можно применять для анализа кода драйвера сразу же после компиляции исходного кода. Для этого не требуется ни выполнять компоновку, ни исполнять код. Это обстоятельство позволяет PREfast выявить ошибочные предположения и ошибки на ранних этапах разработки, когда их легче исправить и когда они обычно оказывают меньшее влияние на график разработки, прежде чем они распространяются по всей программе.

### Внимание!

Инструмент PREfast for Drivers лицензирован только для разработки драйверов. Его не следует применять для тестирования приложений пользовательского режима.

## PREfast и инструмент Code Analysis для Visual Studio

Инструмент PREfast for Drivers содержит компонент, который выявляет основные ошибки в коде программ на языках C и C++, и специализированный модуль для выявления ошибок в коде драйверов режима ядра. Для краткости, в этой главе вместо полного имени PREfast for Drivers употребляется просто PREfast.

Пользователи Visual Studio, может быть, уже имели возможность работать с PREfast. Инструмент для анализа кода на языке C/C++ Code Analysis в Microsoft Visual Studio Team System, Team Edition for Developers имеет ту же функциональность, что и опция /analyze инструмента PREfast, но без специальной функциональности для тестирования драйверов.

## Как работает PREfast

PREfast перехватывает вызов утилиты Build стандартного компилятора cl (cl.exe) и запускает компилятор перехвата, который анализирует исходный код и создает файл журнала протоколирования сообщений об ошибках и предупреждениях. PREfast эмулирует выполнение возможных ветвей кода по одной функции за раз, включая ветви кода, которые редко исполняются при реальной работе драйвера. Каждая возможная ветвь кода проверяется на соответствие правилам, которые определяют возможные ошибки или некачественно написанный код; предупреждения, вызванные кодом, возможно нарушающим эти правила, заносятся в журнал.

Например, PREfast может выявить неинициализированные переменные, которые могут быть использованы в дальнейшем коде, такие как, например, переменные, инициализируемые внутри цикла. Если цикл исполняется нулевое количество итераций, то переменная остается неинициализированной, что создает потенциально серьезную проблему, которую нужно исправить. Если PREfast не может исключить возможность существования ветви кода, в которой может возникнуть такая ситуация, то он выдает предупреждение.

### Примечание

Чтобы повысить производительность, PREfast ограничивает число проверяемых ветвей кода максимальным числом по умолчанию. Максимальное число ветвей кода, которые PREfast может проверить, можно увеличить с помощью опции командной строки /maxpaths.

## Какие ошибки может выявлять PREfast

PREfast может выявлять несколько важных типов потенциальных ошибок в коде сразу же после его компиляции.

- ◆ **Ошибки памяти.** Возможные утечки памяти, разыменованные нулевые указатели, обращения к неиницированной памяти, чрезмерное использование стека режима ядра, а также неправильное использование тегов пулов.
- ◆ **Ошибки ресурсов.** Неосвобождение таких ресурсов, как блокировки, удерживание функцией ресурсов, когда она не должна этого делать, а также неиспользование функций ресурсов, когда они должны использоваться.
- ◆ **Неправильное использование функций.** Возможно неправильное использование определенных функций, подозрительных аргументов функций, возможное несовпадение типов в функциях, которые не выполняют строгой проверки типов, использование определенных устарелых функций, а также вызовы функций на возможно неправильных уровнях IRQL.
- ◆ **Ошибки состояния плавающей запятой.** Невыполнение защиты состояния плавающей запятой аппаратного обеспечения и попытка восстановить состояние плавающей запятой, сохраненного ранее на другом уровне IRQL.
- ◆ **Нарушение правил предшествования.** Код, который может работать не так, как этого хотел программист, по причине правил предшествования языка C.
- ◆ **Некорректные примеры кодирования в режиме ядра.** Неадекватные примеры кодирования, которые могут вызвать ошибки, такие как, например, модифицирование непрозрачной структуры списка MDL, невыполнение проверки значения переменной, установленного вызванной функцией, использование функций для манипуляции строками из библиотеки времени исполнения C, вместо безопасных строковых функций, определенных в файле Ntstrsafe.h, а также неправильное использование страничных сегментов кода.
- ◆ **Неадекватные примеры кодирования, специфичные для драйвера.** Неадекватные специфичные для драйвера операции, которые часто являются источником ошибок в драйверах режима ядра, такие как, например, копирование всего пакета IRP без модифицирующих элементов или сохранение указателя на строковый или структурный аргумент, вместо копирования аргумента в процедуру DriverEntry.

### Внимание!

PREfast очень эффективно выявляет многие ошибки, которые трудно обнаружить другими способами, и обычно докладывает об ошибках способом, облегчающим их исправление. Это способствует высвобождению тестовых ресурсов для концентрирования на выявлении и исправлении более глубоких и серьезных ошибок. Но PREfast не выявляет все возможные ошибки или даже все возможные случаи ошибок, которые он предназначен обнаруживать. Поэтому успешное выполнение тестирования этим инструментом не обязательно означает, что в коде больше нет ошибок. Поэтому обязательно всесторонне протестируйте свой код с помощью всех имеющихся инструментов, включая верификаторы Driver Verifier и Static Driver Verifier. Обзор различных инструментов для тестирования драйверов приводится в главе 21.

## Использование PREfast

Инструмент PREfast можно применять для тестирования как драйверов режима ядра, так и других компонентов режима ядра. Также с его помощью можно анализировать драйверы пользовательского режима. PREfast устанавливается вместе с набором разработчика драйверов WDK, поэтому для его установки не требуется выполнять никаких дополнительных действий.

По умолчанию PREfast анализирует код в соответствии с правилами для драйверов режима ядра. Для анализа драйверов пользовательского режима необходимо установить режим анализа `_user_driver`, как описывается в разд. "Задание режима анализа для PREfast" далее в этой главе, или же просто игнорировать все предупреждения, относящиеся к режиму ядра.

В этом разделе дается короткое введение в использование командной строки PREfast и средства просмотра журнала протоколирования дефектов. Если вы уже знаете, как работать с PREfast, то можете пропустить этот раздел.

### Примечание

Воспользуйтесь в полном объеме возможностями компилятора для проверки ошибок, компилируя исходный код с переключателями `/W4` и `/WX` компилятора в добавление к использованию PREfast. PREfast не активирует переключатель `/W4`, хотя область ошибок, выявляемых с помощью `/W4`, частично перекрывается с областью ошибок, выявляемых PREfast. Большинство этих ошибок имеют отношение к неинициализированным переменным.

## Задание режима анализа для PREfast

Набор правил, используемый PREfast для анализа кода, определяется режимом анализа PREfast. Аннотация режима анализа, определенная в файле `%wdk%\inc\ddk\driverspecs.h`, указывает PREfast, является ли данное тело кода кодом режима ядра или кодом пользовательского режима, а также является ли код драйвером. Эта аннотация применима ко всему файлу исходного кода.

Режим анализа может определяться одной из следующих аннотаций:

- ◆ `_kernel_driver` — для кода драйверов режима ядра, режим анализа по умолчанию;
- ◆ `_kernel_code` — для кода режима ядра, не являющегося драйвером;
- ◆ `_user_driver` — для кода драйверов пользовательского режима;
- ◆ `_user_code` — для кода пользовательского режима, не являющегося драйвером.

Если режим анализа `_kernel_driver` является неправильным для определенного драйвера, то необходимо вставить аннотацию для требуемого режима анализа в файл исходного кода или в соответствующий заголовочный файл сразу же после подключения релевантного заголовка и перед телами функций. Заголовочные файлы `Ntddk.h` и `Wdm.h` включают в себя заголовочный файл `driverspecs.h`, поэтому данную аннотацию можно вставить в любое место после включения файлов `Ntddk.h` или `Wdm.h`.

## Как запустить PREfast

Чтобы запустить PREfast на исполнение в окне среды сборки, введите команду `prefast`, а после нее — обычную команду сборки.

При исполнении команды `prefast`, PREfast перехватывает вызов компилятора, анализирует код, который необходимо скомпилировать, и записывает результаты анализа в файл журнала в формате XML. PREfast обрабатывает по отдельности каждую функцию исходного кода. Он создает один объединенный журнал для всех файлов, проверяемых в одном проходе, и удаляет повторяющиеся ошибки и предупреждения, генерируемые заголовочными файлами. После этого PREfast вызывает стандартный компилятор для выполнения обычной сборки. Полученные в результате объектные файлы такие же, как и создаваемые обычной командой `build`.

### Полезная информация

PREfast предназначен для анализа 32- или 64-битного кода для систем архитектуры x64. При запуске PREfast соответствующая версия инструмента указывается средой сборки WDK. Анализ кода систем типа Itanium можно выполнять двумя способами. При первом способе делается копия кода, которая модифицируется необходимым образом для выполнения сборки в среде сборке x64. При втором способе применяется условная компиляция под архитектуру x64. После этого PREfast запускается в среде сборки x64 на системе с x64-архитектурой.

#### Чтобы запустить PREfast:

1. Откройте окно среды сборки.
2. С помощью команды `cd` установите каталог по умолчанию, требуемый для выполнения сборки исходного кода.  
Например, чтобы выполнить сборку драйвера, для каталога по умолчанию следует указать каталог, содержащий файл `sources` или `dirs`.
3. Введите в командную строку команду

```
prefast build
```

А после нее все параметры для утилиты Build, требуемые для выполнения сборки вашего драйвера. Далее приводится пример такой команды:

```
prefast build -cZ
```

PREfast анализирует код, который необходимо скомпилировать, и записывает результаты анализа в файл журнала в формате XML. Файл журнала по умолчанию называется `Defects.xml` и сохраняется в папке `%wdk%\tools\pdf`. Чтобы сохранить файл журнала в другую папку, используйте с командой `prefast` переключатель `/LOG=`.

## Сборка примеров PREfast

Вместе с PREfast устанавливается папка с примерами исходного кода, содержащего преднамеренные ошибки, чтобы активировать различные предупреждения PREfast. С помощью этих примеров можно проверить правильность установки PREfast на вашу систему, а также поэкспериментировать со средством просмотра журнала протоколирования дефектов. В подпапке `\fail_driver` содержится исходный код драйверов, иллюстрирующий специфичные для драйверов правила более подробно.

Для сравнения с кодом, активирующими предупреждения, в файле `Boundsexamples.cpp` находится несколько функций, не содержащих ошибок и, соответственно, не активирующих никаких предупреждений PREfast. Имена таких функций содержат строку `_ok`.

## Полезная информация

Прежде чем компилировать или модифицировать образец драйвера WDK, скопируйте файлы данного образца в другую папку и впоследствии работайте с этими копиями. Таким образом, оригинальные образцы будут сохранены для дальнейшего использования.

### Для выполнения сборки примеров PREfast:

1. Откройте окно среды сборки.
2. Задайте каталог Samples PREfast в качестве каталога по умолчанию.

Например, если набор разработчика WDK установлен в папку C:\winddk и вы хотите выполнять сборку примеров для PREfast, следуя правилам для драйверов, введите следующую команду в командную строку:

```
cd C:\winddk\tools\pfd\samples
```

3. Для сборки примеров введите команду, подобную следующей:

```
prefast build -cz
```

В листинге 23.1 показаны результаты вывода в командном окне результатов выполнения команды для сборки примеров PREfast. Содержащиеся в нем сообщения об ошибках отражают ошибки в примерах. (Здесь и далее перевод некоторых сообщений листингов за-комментирован и выделен курсивом.)

#### Листинг 23.1. Вывод в командном окне результатов сборки примеров PREfast

```
C:\WINDDK\tools\pfd\samples>prefast build -cz
-----
Microsoft (R) PREfast Version S.O.xxxxx.
Copyright (C) Microsoft Corporation. All rights reserved.
-----
BUILD: Compile and Link for x86
BUILD: Start time: Mon Dec 04 14:37:10 2006
BUILD: Examining c:\winddk\tools\pfd\samples directory for files to compile

c:\winddk\tools\pfd\samples
BUILD: Compiling c:\winddk\tools\pfd\samples directory
__NT_TARCET_VERSION SET TO WINXP
Compiling - bounds-examples.cpp
Compiling - pft-example1.cpp
Compiling - pft-example2.cpp
Compiling - pft-example3.cpp
Compiling - precedence-examples.cpp
Compiling - hresult-examples.cpp
Compiling - drivers-examples.cpp
Compiling - bounds-examples.cpp
Compiling - pft-example1.cpp
Compiling - pft-example2.cpp
Compiling - pft-example3.cpp
Compiling - precedence-examples.cpp
Compiling - hresult-examples.cpp
Compiling - drivers-examples.cpp
Compiling - generating code...
Building Library - objchk_wxp_x86\i386\prefastexamples.lib
```

```
BUILD: Finish time: Mon Dec 04 14:37:19 2006
BUILD: Done
16 files compiled
1 library built
-----
Removing duplicate defects from the log...
// Удаление из журнала повторяющихся ошибок...
-----
PREfast reported 31 defects during execution of the command.
// При исполнении команды PREfast доложил о 31 ошибке.
-----
Enter PREFAST LIST to list the defect log as text within the console.
// Введите команду PREFAST LIST, чтобы вывести журнал ошибок в окне
// консоли в текстовом формате.
Enter PREFAST VIEW to display the defect log user interface.
// Введите команду PREFAST VIEW, чтобы вывести журнал ошибок в
// окне утилиты для просмотра журнала ошибок.
```

## Вывод на экран результатов анализа PREfast

Результаты выполнения анализов PREfast можно вывести для просмотра одним из следующих способов:

- ◆ чтобы просмотреть содержимое файла журнала в утилите просмотра PREfast, выполните команду `prefast view`;
- ◆ чтобы вывести содержимое файла журнала в командном окне среды сборки, выполните команду `prefast list`.

## Утилита просмотра журнала дефектов, обнаруженных PREfast

Утилита для просмотра журнала дефектов, обнаруженных PREfast, оснащена графическим пользовательским интерфейсом. С ее помощью можно просматривать вывод PREfast, накладывать фильтры на этот вывод, чтобы показывать или скрывать определенные сообщения и просматривать аннотированный исходный код, чтобы увидеть путь анализируемого кода, который сгенерировал определенное предупреждение.

### Чтобы вывести результаты работы PREfast в утилите просмотра:

1. Обработайте исходный код инструментом PREfast, как было описано ранее в этом разделе.
2. В командном окне введите следующую команду:

```
prefast view
```

PREfast должен вывести журнал дефектов в окне **Message List**.

### Окно Message List

На рис. 23.1 показан пример вывода в окне **Message List** (Список сообщений) неотфильтрованных результатов выполнения сборки примеров PREfast. Номер версии вверху окна обозначает версию PREfast, которая выводит на экран содержимое журнала.

В окне **Message List** можно выполнять следующие действия:

- ◆ отсортировать сообщения по описанию ошибки, номеру ошибки, файлу исходного кода ошибки и функции, в которой произошла ошибка. Для этого нужно щелкнуть по заго-

ловку соответствующего столбца — **Description** (Описание), **Warning** (Предупреждение), **Source Location** (Размещение источника) или **In Function** (В функции);

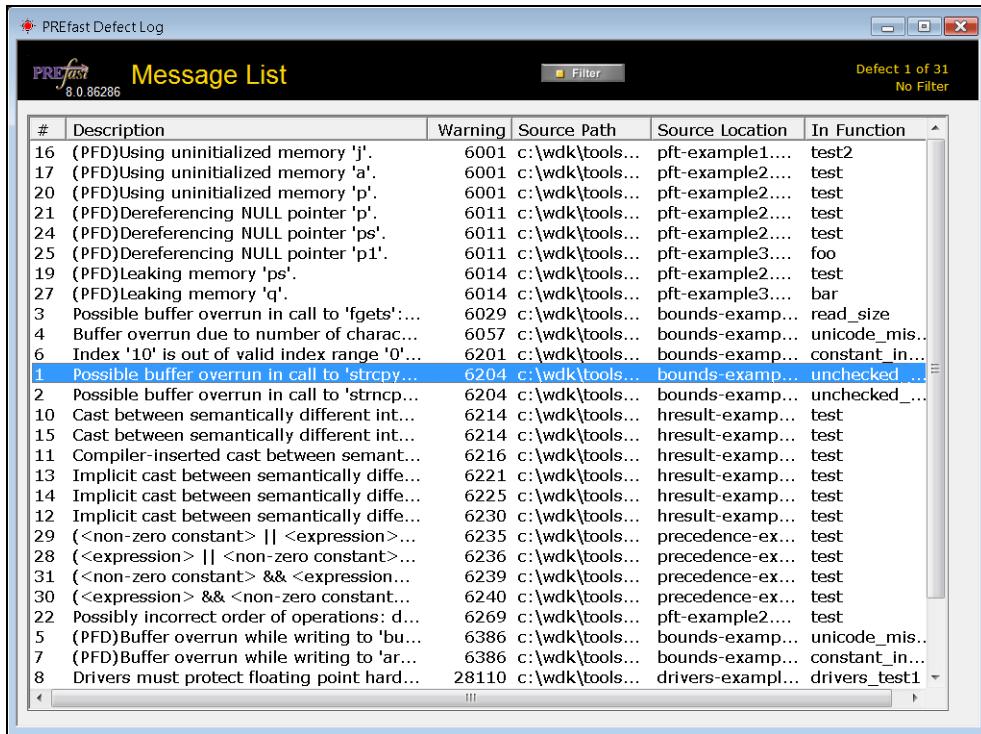


Рис. 23.1. Окно Message List PREfast

- ◆ выполнив двойной щелчок по сообщению, открыть окно **View Annotated Source** (Просмотр аннотированного исходного кода) для просмотра исходного кода, сгенерировавшего данное сообщение;
- ◆ щелкнув кнопку **Filter** (Фильтр), вывести отфильтрованное представление сообщений, в котором можно выбрать из списка предопределенных фильтров необходимый фильтр, чтобы показывать или скрывать индивидуальные сообщения.

### Окно View Annotated Source

Как только что было сказано, двойной щелчок мышью по какому-либо сообщению в окне **Message List** открывает окно **View Annotated Source**. Это окно показано на рис. 23.2. В окне **View Annotated Source** выводится аннотированный исходный код с ошибкой, которая вызвала данное сообщение, с несколькими строчками кода до и после ошибки для контекста.

В окне **View Annotated Source** можно выполнять следующие действия:

- ◆ щелкнув кнопку **Prev** или **Next**, вывести аннотированный код для других сообщений или возвратиться в окно **Message List**, нажав кнопку **Msg List**;
- ◆ под заголовком **View...** нажать кнопку **Show Entire File**, чтобы вывести аннотированный весь исходный код файла, содержащего ошибку;

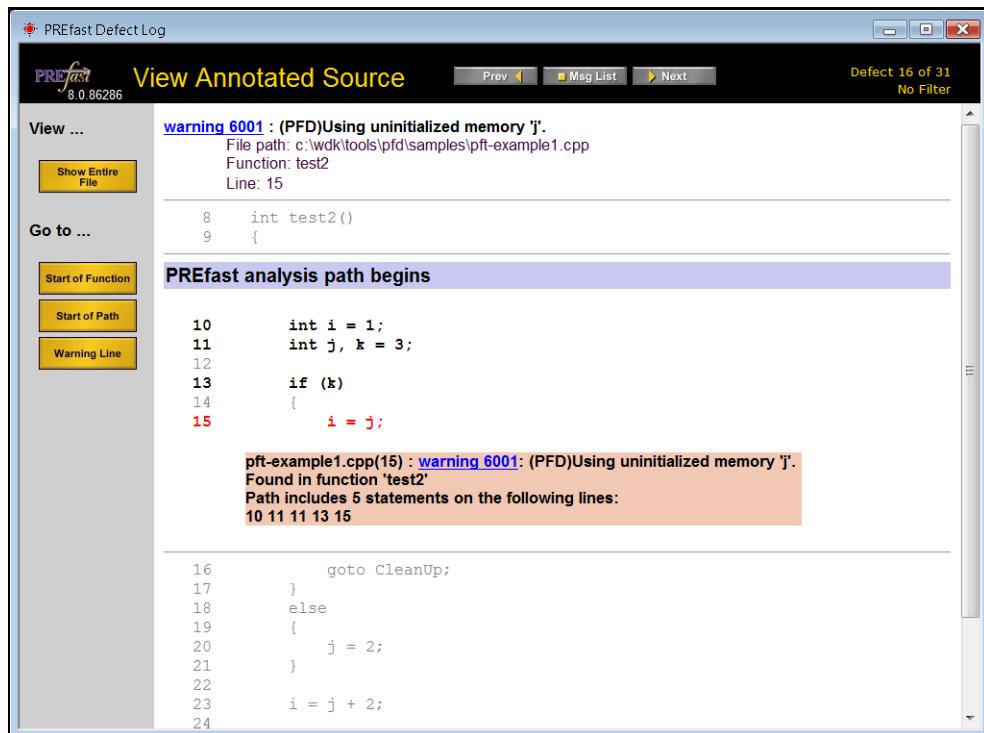


Рис. 23.2. Окно View Annotated Source

- ◆ вывести для просмотра документацию PREfast, в которой подробно описывается данная проблема. Для этого необходимо щелкнуть номер предупреждения;
- ◆ под заголовком **Go to**:
  - перейти в область просмотра к началу функции, щелкнув кнопку **Start of Function**;
  - перейти в область просмотра к началу пути анализа PREfast, нажав кнопку **Start of Path**;
  - нажав кнопку **Warning Line**, перейти в область просмотра к строке кода, которая вызвала предупреждения.

### Полезная информация

Щелчок по номеру предупреждения выводит страницу справки PREfast с подробной информацией о данном предупреждении. В утилите просмотра PREfast текст "warning *nnnn*" (*nnnn* — номер предупреждения) является ссылкой на соответствующую страницу в справке для PREfast for Drivers в документации набора разработчика WDK. Для многих предупреждений документация предоставляет значительную информацию, помогающую постижению сущности предупреждения, и часто предлагает возможное решение проблемы. Если вы не понимаете значения определенного предупреждения, обратитесь к документации — это может сэкономить вам много времени.

### Окно Message List в представлении Filter

Если щелкнуть кнопку **Filter** в окне **Message List**, то поверх списка сообщений, сгенерированных при сборке, будет выведен список сообщений, которые можно фильтровать (рис. 23.3).

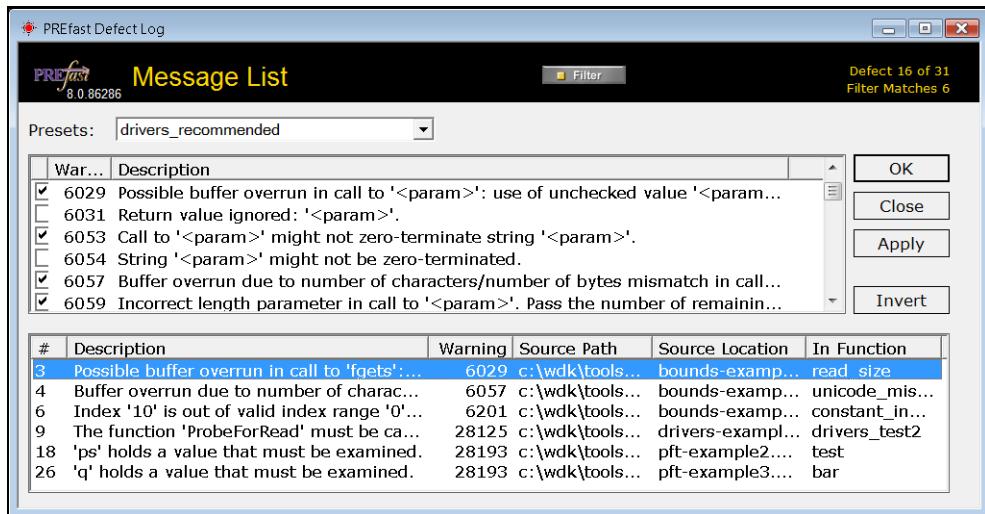


Рис. 23.3. Окно Message List в представлении Filter

В представлении **Filter** окна **Message List** можно выполнять следующие операции:

- ◆ выбрать предопределенные фильтры для вывода только сообщений, отвечающих критериям выбранных фильтров;
- ◆ в панели фильтров сообщений сбросить флажок сообщения или выбрать сообщение и нажать кнопку **Invert**, чтобы не выводить данное сообщение в списке сообщений;
- ◆ щелкнуть кнопку **Apply** для обновления списка сообщений, чтобы в нем показывались только сообщения, выбранные в панели фильтров сообщений;
- ◆ щелкнуть кнопку **Filter** опять, чтобы убрать панель фильтров. В окне **Message List** теперь будут отображаться только сообщения, выбранные в панели фильтров сообщений.

В представлении **Filter** можно также выполнить двойной щелчок мышью по сообщению, чтобы вывести окно **View Annotated Code**, когда панель фильтров не выведена.

### Полезные советы для фильтрации результатов PREFast

Фильтрация сообщений не означает, что PREFast не будет обнаруживать соответствующих ошибок. Таким образом, список сообщений просто прореживается от ненужных вам сообщений, чтобы легче было разобраться с сообщениями интересующего вас типа. После исправления ошибок, соответствующих отфильтрованным сообщениям, всегда нужно обработать исходный код инструментом PREFast опять и просмотреть сообщения с установленными новыми фильтрами. Таким образом, вы сможете увидеть и исправить другие, менее критические ошибки.

### Воспользуйтесь предопределенными фильтрами

Фильтр **drivers\_recommended** отфильтровывает сообщения для серьезных ошибок, как в коде общего назначения, так и в драйверном коде. Эти сообщения идентифицируют ситуации, которые обычно оказываются настоящими ошибками, а не такими, которые называются ложными обнаружениями или шумом. Фильтр **drivers\_only** выводит сообщение только для ошибок, специфичных только для драйверов. Если у вас не хватает времени, чтобы испра-

вить все ошибки, обнаруженные PREfast в драйвере, то воспользуйтесь одним из этих предопределенных фильтров, чтобы выделить наиболее серьезные ошибки и концентрироваться на их исправлении.

### Скрытие отдельных сообщений

Для скрытия отдельных сообщений могут быть разнообразные причины: команда разработчиков может полагать, что риск, ассоциированный с сообщением, является приемлемым или что уровень шума неприемлемо высокий, цикл отправки продукта может быть слишком коротким и позволяет исправить только наиболее критические ошибки, или же сообщения могут просто не иметь значения для вашего проекта.

Например, определенные предупреждения PREfast, относящиеся к драйверам режима ядра, также активируются драйверами пользовательского режима. При тестировании драйверов пользовательского режима может быть необходимым скрыть сообщения драйверов режима ядра. Например, такие, как следующие:

```
Warning 28110: Drivers must protect floating point hardware state.  
See use of float <expression>  
Warning 28111: The IRQL where the floating point state was saved  
does not match the current IRQL (for this restore operation)  
Warning 28146: Kernel mode drivers should use ntstrsafe.h,  
not strsafe.h
```

-----  
Предупреждение 28100: драйверы должны предохранять аппаратное  
состояние плавающей точки.

См. материалы по использованию плавающего <выражение>

Предупреждение 28111: уровень IRQL, на котором было сохранено  
состояние плавающей запятой, не совпадает с текущим уровнем IRQL  
для этой операции восстановления).

Предупреждение 28146: драйверы режима ядра должны использовать  
файл ntstrsafe.h, а не файл strsafe.h

Чтобы не выводить отдельное сообщение, необходимобросить его флагок в панели отображения фильтров сообщений, как показано на рис. 23.3.

### Вывод журнала дефектов PREfast в текстовом виде

Содержимое журнала дефектов можно вывести в текстовом виде в командном окне среды сборки с помощью команды prefast list. Эта команда может быть полезной, если вам достаточно только краткого списка ошибок, и выводить весь аннотированный исходный код нет надобности. Например, если вы просто хотите просмотреть эффект исправления ошибок, обнаруженных PREfast в предыдущем проходе. Команда prefast list выводит такую же информацию, как и утилита просмотра PREfast, в формате, пригодном для вставки информации в файлы или отчеты об ошибках.

#### Чтобы вывести результаты анализа PREfast в текстовом формате:

1. Обработайте исходный код инструментом PREfast, как было описано ранее в этом разделе.
2. В командном окне введите следующую команду:

```
prefast list
```

PREfast должен вывести список сообщений в командном окне.

В листинге 23.2 приводится пример текстового вывода первых нескольких сообщений, сгенерированных при сборке примеров PREfast.

#### Листинг 23.2. Текстовый список сообщений сборки примеров PREfast

```
C:\WINDDK\tools\pfd\samples>prefast list
-----
Microsoft (R) PREfast Version 8.0.58804.
Copyright (C) Microsoft Corporation. All rights reserved.
-----
Contents of defect log:
C:\Documents and Settings\<username>\ApplicationData
    \Microsoft\PFD\defects.xml
-----
c:\winddk\tools\pfd\samples\bounds-examples.cpp
(45): warning 6029: Possible buffer overrun in call to 'fgets':
use of unchecked value 'line_length'
// Возможное переполнение буфера в вызове 'fgets': использование
// непроверенного значения 'line_length'
FUNCTION: read_size (40)
c:\winddk\tools\pfd\samples\bounds-examples.cpp (54): warning 6057:
Buffer overrun due to number of characters/number of bytes mismatch
in call to 'wcsncpy'
// Переполнение буфера, вызванное несоответствием числа
// символов/байтов в вызове 'wcsncpy'
    FUNCTION: unicode_misuse (51)
c:\winddk\tools\pfd\samples\bounds-examples.cpp (62): warning 6201:
Index '10' is out of valid index range '0' to '9' for possibly
stack allocated buffer 'arr'
// Индекс '10' вне пределов диапазона действительных индексов
// '0' to '9' для буфера 'arr', возможно выделенного в стеке.
    FUNCTION: constant_index (59)
c:\winddk\tools\pfd\samples\drivers-examples.cpp (23): warning 28125:
The function 'ProbeForRead' must be called from within a try/except
block: The requirement might be conditional.
// Функция 'ProbeForRead' должна вызываться из блока try/except:
// Это требование может быть условным.
    FUNCTION: drivers_test2 (21)
```

## Примеры результатов анализа PREfast

В этом разделе приводятся несколько простых примеров исходного кода и описываются решения распространенных ошибок, которые PREfast может выявить в исходном коде. Эти примеры показываются без аннотаций исходного кода, чтобы можно было видеть, что именно PREfast выявляет в исходном коде без аннотаций.

### Пример 1: неинициализированные переменные и нулевые указатели

Примеры PREfast в наборе WDK содержат преднамеренные ошибки, а также написаны с применением некачественных приемов программирования, чтобы показать, как PREfast реагирует на ошибки и некачественное программирование. В примере %wdk%\tools\pfd\samples\pft-example2.cpp тестовая функция активирует несколько сообщений PREfast, связанных с

неинициализированными переменными и нулевыми указателями. Хотя ошибки в этом при-  
мере можно легко видеть, просто просматривая код, они иллюстрируют ошибки, которые  
могут быть не так легко заметны в более сложном коде без специального средства, такого  
как PREfast.

На рис. 23.4 показан путь анализа PREfast для одного из предупреждений, связанного с пе-  
ременной-указателем `p` в функции `test`. Операторы в пути анализа выделены жирным шриф-  
том.

```

12     void test()
13     {
14         int *p, a;
15         S *ps, c;
16
17         if (a)
18         {
19             p = &a;
20         }
21         else
22         {
23             ps = (struct S*)malloc(sizeof(struct S));
24         }
25
26         if (p)
27         {
28             ps = &c;
29         }
30
31         *p;
pft-example2.cpp(31) : warning 6011: Dereferencing NULL pointer 'p'.
Found in function 'test'
Path includes 8 statements on the following lines:
14 14 15 15 17 23 26 31
32         a = (((ps))->a;
33
34         return;
35     }

```

Рис. 23.4. Пример неинициализированной переменной и нулевых указателей

В этой функции объявляется, но не инициализируется несколько переменных, и поэтому в  
функции не выполняется должным образом ветвление. В данной ветви кода проверка усло-  
вия в строке 17 завершается неудачей, т. к. переменная `a` не была инициализирована, о чем  
сообщается в другом предупреждающем сообщении, не показанном на рис. 23.4. Поэтому  
строка кода 19 не исполняется, оставляя переменную `p` неинициализированной.

Хотя переменная `p` проверяется в строке кода 26, обработка ошибки проверки не преду-  
смотрена, и исполнение продолжается в строке 31. Здесь переменная `p` разыменовывается,  
что активирует сообщение PREfast о том, что переменная `p` может быть `NULL`. Нулевой указа-  
тель может также активировать предупреждение о неинициализированных переменных, как  
это происходит с переменной `p` в рассматриваемой функции. Чтобы устранить эту ошибку,  
следует добавить логику для предотвращения разыменовывания переменной `p`, если она  
`NULL`.

## Пример 2: неявный порядок вычислений

Код, основанный на неявном порядке вычислений, может содержать трудновыявляемые  
ошибки. PREfast выявляет ситуации, где неявный порядок вычислений может дать результа-  
ты, отличающиеся от предполагаемых программистом.

Простой пример такого кода показан на рис. 23.5.

Согласно правилам предшествования операторов языка C, выражение `(a & b == c)` интерпретируется как `(a & (b == c))`, т. к. логический оператор проверки на равенство `==` имеет более высокий приоритет, чем побитовый оператор `AND` (`&`). Поэтому данная функция сравнивает переменную `b` с переменной `c`, после чего маскирует результат переменной `a`, в результате выполняя проверку, является ли значение переменной `a` четным или нечетным. Если это является желаемым результатом, то функция написана правильно в том виде, в каком она есть. Тем не менее применение скобок сделало бы намерения программиста более ясными, так же, как и соответствующий комментарий к этому фрагменту кода.

```

10     int unclearIntent(int a, int b, int c)
11     {
PREfast analysis path begins
12     if (a & b == c) return 1;
      pfdsts7.c(12) : warning 6281: Incorrect order of operations: relational
      operators have higher precedence than bitwise operators.
      Found in function 'unclearIntent'
12
13     return 0;
14 }
```

Рис. 23.5. Пример неявного порядка вычислений

Но если намерением программиста было сначала выполнить операцию маскирования переменной `a` переменной `b` и сравнить результат с переменной `c`, то тогда функция написана неправильно. Для желаемого результата необходимо применить скобки, чтобы заставить принудительное вычисление выражения как `((a & b) == c)`.

### Пример 3: вызов функции на неправильном уровне IRQL

Уровень IRQL, на котором исполняется драйверная функция, определяет, какие функции режима ядра она может вызывать, а также, может ли она обращаться к страничной памяти, использовать объекты диспетчера ядра или выполнять другие действия, которые могут вызвать страничную ошибку. Например, для некоторых функций интерфейса DDI требуется, чтобы вызывающий клиент исполнялся на уровне `IRQL = DISPATCH_LEVEL`, в то время как безопасный вызов других функций не может быть выполнен на уровне IRQL выше, чем `PASSIVE_LEVEL`.

Многие функции интерфейса DDI, которые должны вызывать драйверы, зависят, иногда тонким образом, от уровня IRQL. Например, если драйвер устанавливает флаг по результатам сравнения двух строк, который сохраняется в структуре с защищенным доступом, у программиста может возникнуть соблазн написать код для этого подобно примеру, показанному на рис. 23.6. В данном примере вызов процедуры `ExAcquireFastMutex` сбрасывает текущий уровень IRQL функции до `APC_LEVEL`. Но функция `RtlCompareUnicodeString` должна вызываться на уровне `IRQL = PASSIVE_LEVEL`, поэтому PREfast выдает предупреждение.

Обратите внимание на предложение "IRQL was last set to 1 at line 14" ("Последний раз уровень IRQL = 1 был установлен в строке кода 14"). В этом коротком примере данную ошибку, вероятно, и можно было бы увидеть самостоятельно, но для обнаружения ее в более длинной и сложной функции, информация этого рода, предоставляемая PREfast, может оказаться очень полезной.

```

11     void IsFlagSet(
12         IN PUNICODE_STRING s)
13     {
PREfast analysis path begins
14             ExAcquireFastMutex(&mutex);
15
16             if (RtlCompareUnicodeString(s, &t, TRUE) == 0) {
IRQL.c(16): warning 28121: The function 'RtlCompareUnicodeString' is not
permitted to be called at the current IRQ level. The current level is too high:
IRQL was last set to 1 at line 14. The level might have been inferred from the
function signature.
17             Found in function 'IsFlagSet'
18             Path includes 2 statements on the following lines:
19             14 16
20             MyGlobal.FlagIsSet = 1;
21         }
ExReleaseFastMutex(&mutex);
}

```

Рис. 23.6. Пример вызова функции на неправильном уровне IRQL

Чтобы исправить данную ошибку, следует переместить вызов процедуры `RtlCompareUnicodeString` до вызова процедуры `RtlCompareUnicodeString`, а потом проверить результат после получения мьютекса. Пример исправленного таким образом кода показан в листинге 23.3.

#### Листинг 23.3. Исправленный код из рис. 23.6

```

void IsFlagSet(
    IN PUNICODE_STRING s)
{
    int tmp = 0;
    if (RtlCompareUnicodeString(s, &t, TRUE) == 0) {
        tmp = 1;
    }
    ExAcquireFastMutex(&mutex);
    if (tmp) {
        MyGlobal.FlagIsSet = 1;
    }
    ExReleaseFastMutex(&mutex);
}

```

#### Пример 4: действительная ошибка, но указанная в неправильном месте

PREfast часто выдает в одной части кода извещения об ошибке, которая в действительности была вызвана кодом в другом месте. Хорошим примером такого извещения может служить вызов функции на неправильном уровне IRQL. Когда PREfast анализирует ветвь кода, он пытается предположить диапазон значений уровня IRQL, в котором могла бы исполняться функция, и выявить любые несоответствия, как показано в данном примере. PREfast действует исходя из того, что все изменения уровня IRQL являются результатом намерения программиста. Если же следующая функция вызывается на неправильном уровне IRQL, то PREfast выдает предупреждение о вызове функции на неправильном уровне IRQL, а не о предыдущем изменении этого уровня.

## Полезная информация

Аннотации для IRQL, такие как `_drv_requiresIRQL`, помогают PREfast делать более точные предположения о диапазоне уровней IRQL, в котором должна исполняться функция. Дополнительную информацию по этому вопросу см. в разд. "Аннотации IRQL" далее в этой главе.

Например, допустим, что функция `Y` должна вызываться на уровне `IRQL = DISPATCH_LEVEL`. Эта функция вызывает две процедуры интерфейса DDI, имеющие особые требования к уровню IRQL. А именно, процедура `KeDelayExecutionThread` должна вызываться на уровне `IRQL = APC_LEVEL`, а процедура `KeReleaseSpinLockFromDpcLevel` — на уровне `IRQL = DISPATCH_LEVEL`.

Когда PREfast начинает анализировать функцию `Y`, вызов процедуры `KeAcquireSpinLockAtDpcLevel` заставляет его полагать, что код, должно быть, исполняется на уровне `IRQL = DISPATCH_LEVEL`<sup>1</sup> и поэтому он выдает предупреждение, что процедура `KeDelayExecutionThread` вызывается на слишком высоком уровне IRQL (рис. 23.7).

```

17 void Y(void)
18 //This routine will be always called at DISPATCH_LEVEL
19 {
PREfast analysis path begins
20     LARGE_INTEGER SleepTime;
21     KeAcquireSpinLockAtDpcLevel(&spinLock);
22
23     if(some_condition) {
24         //Driver performs I/O to the hardware and decides
25         //to get a response or check the state.
26
27         KeDelayExecutionThread(KernelMode, FALSE, &SleepTime);
xy.c(27) : warning 28123: The function KeDelayExecutionThread is not
permitted to be called at a high IRQ level. Prior function calls are
inconsistent with this constraint: It may be that the error is actually in
some prior call that limited the range. Minimum legal IRQL was last
set to 2 at line 21.
        Found in function 'Y'
        Path includes 4 statements on the following lines:
        20 21 23 27
28     }
29     KeReleaseSpinLockFromDpcLevel(&spinLock);
30 }
```

Рис. 23.7. Пример действительной ошибки, указанной в неправильном месте

Продолжая анализировать функцию `Y`, PREfast предполагает, что теперь код должен исполняться на уровне `IRQL = APC_LEVEL`, т. к. была вызвана процедура `KeDelayExecutionThread`. Потом он доходит до процедуры `KeReleaseSpinLockFromDpcLevel`, которая должна вызываться на уровне `IRQL = DISPATCH_LEVEL`, полагает, что она вызывается на слишком низком уровне IRQL, и выдает следующее предупреждение:

```

xy.c(29) : warning 28122: The function
KeReleaseSpinLockFromDpcLevel is not permitted to be called at
a low IRQ level. Prior function calls are inconsistent with this
constraint: It may be that the error is actually in some prior call that
limited the range. Maximum legal IRQL was last set to 1 at line 27
        Found in function 'Y'
```

<sup>1</sup> При вызове процедуры `KeAcquireSpinLockAtDpcLevel` предполагается, что уровень IRQL был повышен до требуемого перед ее вызовом. — Пер.

Path includes 5 statements on the following lines:  
20 21 23 27 29

ху.с(29) : предупреждение 28122: запрещается вызывать функцию KeReleaseSpinLockFromDpcLevel на низком уровне. Предыдущие вызовы других функций несовместимы с этим ограничением.  
Возможно, что ошибка в действительности находится в одном из предыдущих вызовов функций, который ограничил уровень значений IRQL.  
Последний раз максимальный позволенный уровень IRQL был установлен в 1 в строке кода 27.

Ошибка обнаружена в функции 'Y'

Ветвь содержит 5 операторов в следующих линиях кода:

20 21 23 27 29

Эту ситуацию можно исправить одним из следующих способов.

- ◆ Для короткого периода ожидания (больше одного такта системных часов, но меньше чем несколько инструкций) вместо процедуры KeDelayExecutionThread воспользуйтесь процедурой KeStallExecutionProcessor. Клиенты,зывающие процедуру KeStallExecutionProcessor, могут исполняться на любом уровне IRQL.
- ◆ Поставить в очередь объект таймера для процедуры CustomTimerDpc или для функции обратного вызова события EvtTimerFunc, которая проверяет состояние аппаратуры.

### Пример 5: несоответствие класса типа функции

PREfast относится более строго, чем компилятор, к назначениям функций обратного вызова указателям функций. Для этого он присваивает каждой функции обратного вызова "класс типа".

В PREfast класс типа играет роль типа, но выходит за пределы концепции типа в определении языка С и не имеет отношения к классу в определении языка С++. Когда PREfast обнаруживает или несоответствие класса типа, или тип функции, который не имеет класса типа, он выдает ошибку. PREfast также использует класс типа для выполнения проверок, специфичных для определенного типа функций, при этом не выполняя их по ошибке для функций, которые просто выглядят, как данный тип функций.

Типичная ошибка класса типа функции определена предупреждением 28155:

**28155 — The function being assigned or passed should be a <class1> function. Add the declaration "<class1> <funcname1>" before the current first declaration of <funcname2>.**

**28115 — Назначаемая или передаваемая функция должна быть типа <class1>.**

*Добавьте объявление <class1> <funcname1> перед текущим первым объявлением <funcname2>.*

Это предупреждение говорит о том, что функция, назначаемая определенному указателю функции, не соответствует ожидаемому типу. Примером такой ошибки может быть попытка назначить процедуру Cancel указателю функции StartIo. Компилятор С позволяет делать это, но PREfast — нет. Обычно назначение, вызвавшее предупреждение, выполняется правильно, но функция не принадлежит ни к какому определенному классу функций.

Пример такой ошибки показан на рис. 23.8.

Чтобы исправить эту ошибку, следует добавить DRIVER\_CANCEL MyCancel; перед строкой кода 2124, чтобы дать знать PREfast, что функция MyCancel является функцией отмены. Это пода-

вит предупреждение 28155 и вынудит PREfast проверять функцию на соответствие требованиям процедуры отмены.

```

2124 void MyCancel( struct _DEVICE_OBJECT *DeviceObject,
2125                  struct _IRP *Irp)
2126 {
2127     //...
2128 }
2129
2130     Irp->CancelRoutine = MyCancel;
fun.c(3130) : warning C28155: The function being assigned or passed
should be a DRIVER_CANCEL function: Add the declaration
'DRIVER_CANCEL MyCancel;' before the current first declaration of
MyCancel.
Found in function 'MyCancel'
3130

```

Рис. 23.8. Пример ошибки несоответствия класса типа функции

Дополнительную информацию об объявлении `typedef` функции см. в разд. "Аннотации к объявлению `typedef` функций" и "Аннотации класса типа функций" далее в этой главе.

### Пример 6: неправильный перечислимый тип

Контроль типов, осуществляемый компилятором С, недостаточно строгий, чтобы выявить некорректный перечислимый тип. К сожалению, использование неправильного перечислимого типа в коде драйвера может вызвать проблемы, которые трудно обнаружить и исправить. PREfast обнаруживает несовпадения перечислимых типов в коде драйвера и выдает соответствующее предупреждение о каждом несоответствии. Предупреждения, выдаваемые PREfast о несовпадении типов, несколько отличаются в зависимости от анализируемой функции, но все они указывают на потенциальные проблемы, которые необходимо рассмотреть и исправить.

Например, распространенной ошибкой при вызове процедур семейства KeWaitXXX (KeWaitForSingleObject, KeWaitForMultipleObjects и KeWaitForMutexObject) является транспонирование параметров WaitReason и WaitMode, которые принимают членов перечислений типа KWAIT\_REASON и KPROCESSOR\_MODE соответственно.

Пример предупреждения PREfast для вызова процедуры KeWaitForSingleObject с транспонированными параметрами WaitReason и WaitMode показан на рис. 23.9.

```

2784 status = KeWaitForSingleObject(&event,
2785
2786     KernelMode,
2787     Executive,
2788     FALSE,
2789     NULL
2790 );

```

Рис. 23.9. Пример предупреждения для неправильного перечислимого типа

Такое же самое предупреждение PREfast выдает для аргумента `Executive`, т. к. он не имеет тип `KPROCESSOR_MODE`.

В вызовах процедур семейства `KWaitXxx` наиболее часто передаются член `Executive` перечисления `KWAIT_REASON` и член `KernelMode` перечисления `KPROCESSOR_MODE`. Значение этих обоих членов равно нулю, поэтому они взаимозаменяемые в числовом отношении. Если драйвер транспонирует их в вызове функции, то для каждого параметра происходит несоответствие типа. Без строгой проверки типов компилятор не видит этого несоответствия.

Но если транспонировка происходит, когда эти параметры содержат невзаимозаменяемые члены перечисления, такие как, например `Executive` и `UserMode`, таким образом вынуждая драйвер ожидать в режиме, который не был намерением программиста, то возникает проблема.

В этом примере показано, как PREfast может помочь предотвратить ошибку, если начать применять его на ранних стадиях процесса разработки. Если значения транспонированных параметров ненулевые, то в какой-то момент тестирования эта ошибка в коде выявится. Некоторые разновидности этой ошибки в коде могут вызывать ошибки исполнения, которые выглядят как останов `bugcheck` в ядре или в драйвере, не имеющем отношения к данной ситуации. Это обстоятельство делает очень трудной задачу распознания и обнаружения этой ошибки в драйвере. Но PREfast выявляет эту ошибку, что позволяет сэкономить время на тестирование и отладку, которое в противном случае нужно было бы затратить на решение этой проблемы. Исправление ошибки занимает незначительное время, т. к. PREfast выдает специфическое сообщение об ошибке и помечает область ее нахождения в коде.

## Практики кодирования, улучшающие результаты анализа PREfast

PREfast протоколирует все ошибки, которые он может обнаружить в исходном коде. Если PREfast не обнаруживает в коде каких-либо признаков безопасности, то он полагает, что код небезопасен.

Например, допустим, что PREfast находит ветвь кода, в которой разыменовывается указатель. Если имеется какой-либо повод полагать, что данный указатель может быть `NULL`, например, если ранее в коде выполнялась проверка на `NULL`, но последующий код обращается к указателю небезопасным образом, то PREfast выдает предупреждение о попытке разыменовать нулевой указатель. Но если причин для подозрений, что указатель может когда-либо быть нулевым, не имеется, то PREfast не выдает такого предупреждения.

PREfast может пропустить возможные ошибки, вследствие ложных предположений; такие результаты часто называются ложноотрицательными (*false negative*). В других случаях предупреждения, выдаваемые PREfast, не отображают настоящие ошибки в коде. Такие результаты часто называются ложноположительными (*false positive*) или шумом (*noise*).

Ложноположительными результатами анализов PREfast не следует пренебрегать. Часто они обозначают предположения о том, как код будет действительно исполняться. Например, если переменная инициализируется внутри тела цикла, вы можете знать, что цикл не может исполняться нулевое количество итераций или что переменная надежно инициализируется какой-либо другой функцией. Но предупреждение PREfast о том, что эта переменная может быть неинициализированной, идентифицирует предположение, что данная переменная будет всегда надежно инициализированной, что является ценной информацией.

Результаты анализов PREfast неизбежно будут содержать определенное количество шума, который можно просто игнорировать или подавить. PREfast анализирует код по одной функции за раз, поэтому у него нет информации о глобальном состоянии или работе, которая выполняется вне функции, анализируемой в настоящее время, которые могут повлиять на исполнение данной ветви кода. По этой причине, PREfast часто докладывает ложноположительные результаты, связанные со следующими.

- ◆ Членами структур или другими объектами, не являющимися простыми переменными. Эти объекты могут дезориентировать более аккуратные проверки потока управления.
- ◆ Оберточные функции. Эти компоненты могут вызвать ложноположительные результаты в случаях со многими предупреждениями: об утечках памяти и ресурсов, разыменовании нулевых указателей, обращениях к неинициализированным областям памяти и неправильных типов аргументов. Многие проблемы с оберточными функциями можно решить, применяя аннотации в исходном коде.

В этом разделе описываются методы, с помощью которых можно уменьшить уровень шума в результатах анализов PREfast и, таким образом, улучшить их.

## Предупреждения, указывающие распространенные причины шума и как на них реагировать

Уровень шума в результатах анализов PREfast, вызванного неадекватными методами написания кода, можно уменьшить внесением в код незначительных изменений. Хотя такие изменения могут казаться тривиальными, они могут как подавить шум, так и способствовать облегчению обслуживания кода другими разработчиками. Кроме этого, PREfast часто выдает несколько предупреждений об одной и той же ошибке, но в слегка разных контекстах. Таким образом, внесением одного изменения в код можно устраниТЬ несколько предупреждений.

Следующие типы предупреждений PREfast часто идентифицируют распространенные причины шума в результатах анализов PREfast.

- ◆ **Предупреждения о неинициализированных переменных.**

При любой возможности инициализируйте переменные при их объявлении.

- ◆ **Предупреждения, активирующиеся явной проверкой на STATUS\_SUCCESS.**

Замените явные проверки на STATUS\_SUCCESS макросом NT\_SUCCESS, как показано в следующем фрагменте кода:

```
status = IoAttachDevice();  
if (!NT_SUCCESS(status)) {  
    // Обработать ошибку  
}
```

- ◆ **Многочисленные предупреждения, активируемые одной ошибкой, такие как многократное использование одного и того же нулевого указателя.**

Исправьте ошибку, после чего повторите проверку с PREfast, чтобы получить более короткий список сообщений.

- ◆ **Предупреждения, идентифицирующие предположения.**

Делайте предположения в вашем коде явными, используя утверждения, такие как макрос ASSERT, или аннотацию исходного кода \_analysis\_assume(*выражение*). Например, если

PREfast докладывает возможное применение неинициализированной переменной, которая, как вам известно, была надежно инициализирована в другом месте, то добавьте утверждение, чтобы подтвердить, что ветвь, в которой переменная оставляется неинициализированной, является невозможной, и воспользуйтесь извещением проверочной сборки, в случае неудачного исполнения утверждения.

Подробную информацию об аннотации `_analysis_assume` см. в статье "How to: Specify Additional Code Information" ("Как указывать дополнительную информацию в коде") в MSDN по адресу <http://go.microsoft.com/fwlink/?LinkId=80906>.

◆ **Предупреждения, идентифицирующие ошибки с применением скобок или другие синтаксические некорректности.**

Добавьте необходимые скобки или модифицируйте код иным образом, чтобы сделать ваши намерения явными. Без таких модификаций код может работать не так, как вы намеревались, по причине правил предшествования языка С.

◆ **Предупреждения, которые можно устраниТЬ незначительными перестановками кода.**

Например, если переменная инициализируется внутри цикла, который может исполняться нулевое количество раз, таким образом оставляя переменную неинициализированной, подумайте об изменении кода, чтобы инициализация переменной выполнялась вне цикла, или переделайте цикл так, чтобы он гарантированно исполнялся, по крайней мере, один раз.

◆ **Предупреждения о возможно неправильном использовании указателей функций.**

Используйте объявления функции `typedef` для идентификации типов системных функций обратного вызова. PREfast может воспользоваться этими объявлениями для проверки на правильность использования указателей функций, что как уменьшает уровень шума, так и улучшает аккуратность анализов. Соответствующий пример приводится в разд. "Пример 5: несоответствие типов классов функций" ранее в этой главе.

## **Воздействие вставленного кода ассемблера на результаты анализов PREfast**

Так как PREfast просто игнорирует вставленный код ассемблера, то применение такого кода в обычном исходном коде может предотвратить полный анализ ветви кода и может вызвать как ложноположительные, так и ложноотрицательные результаты анализов. Результатом использования вставленного кода ассемблера также может быть ухудшение переносимости на более новые архитектуры, поддерживаемые Windows.

Чтобы уменьшить воздействие вставленного кода ассемблера на результаты анализов PREfast:

- ◆ используйте сервисные функции, предоставляемые более новыми компиляторами. Например, вместо `_asm int 3` используйте `_debugbreak`. Для получения подробностей обратитесь к документации для используемого вами компилятора;
- ◆ если избежать применения вставленного кода ассемблера является невозможным, поместите его в функцию `_inline` или `_forceinline`, чтобы PREfast мог анализировать остальную часть функции более эффективно.

## Использование директивы *pragma warning* для подавления шума

Если вы определите, что предупреждение PREfast является ложноположительным или просто шумом, исправлять которое нет необходимости, его можно подавить с помощью директивы `#pragma warning`. В отличие от фильтров, которые временно изменяют отображение результатов в журнале дефектов PREfast, директива `#pragma warning` воздействует на анализ кода инструментом PREfast.

Для идентификации предупреждения, которое необходимо подавить, используйте его номер PREfast в директиве `#pragma warning`. Чтобы ограничить эффект директивы строкой кода, вызывающей ложноположительный результат, можно использовать операторы `(push)` и `(pop)`, как показано в следующем коде:

```
#pragma warning (push)
#pragma warning( disable:6001 ) // FLAG is always present in arr
    arr[i+1] = 0;
#pragma warning (pop)
    j++; // Warning 6001: Actual error
```

Для подавления предупреждения в определенной строке кода, и только в данной строке кода, в качестве альтернативы операторам `push` и `pop` можно использовать спецификатор `suppress` сразу же после оператора `#pragma warning`, как показано в следующем примере:

```
#pragma warning( suppress : 6001 )
arr[i+1] = 0; // Warning 6001 is suppressed
j++;          // Warning 6001 is reported
```

Не забывайте, что применение директивы `#pragma warning` является крайней мерой, т. к. она изменяет исходный код таким образом, чтобы PREfast не докладывал об ошибках. Поэтому подумайте о решении просто игнорировать или фильтровать предупреждения PREfast до тех пор, пока не выйдет следующая версия PREfast, которая может давать более аккуратные результаты.

Если же вы все-таки решите подавлять предупреждения PREfast с помощью директивы `#pragma warning`, обязательно закомментируйте код объяснением, почему данное предупреждение подавляется. При установке новой версии PREfastdezактивируйте директивы `#pragma warning` и проверьте код новой версией PREfast, чтобы увидеть, исправит ли она проблему ложноположительных предупреждений.

Дополнительную информацию по теме см. в разделе **Using a Pragma Warning Directive** (Применение директивы `pragma warning`) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80908>.

## Использование аннотаций для устранения шума

Аннотации могут предоставить PREfast информацию о глобальном состоянии или работе, которая выполняется вне функции, анализируемой в настоящее время, которые могут повлиять на выполнение данной ветви кода. Располагая более подробной информацией о предполагаемом использовании аннотированной функции, PREfast может лучше определить, является ли данная ошибка настоящей. Аннотации могут значительно уменьшить число как ложноположительных, так и ложноотрицательных результатов анализов PREfast.

Например, с помощью аннотации `_bcount(size)` можно выразить размер буфера в байтах. Параметр `size` может быть числом, но обычно это имя какого-либо параметра в функции,

подвергающейся аннотации. Хорошим примером использования этой аннотации является ее применение в функции `memset`, как показано в следующем коде:

```
void * memset(
    _out_bcount(s) char *p,
    _in int v,
    _in size_t s
);
```

В данном примере аннотация `_out_bcount(s)` указывает, что содержимое области памяти, указываемой выходным параметром `p`, устанавливается функцией и что количество байтов, которые нужно установить, указывается в `s`. Хоть предоставляемая таким образом информация и известна "всем", компилятор не является одним из этих всех. В исходном коде С нет ничего, что могло бы сказать компилятору, что переменные `p` и `s` связаны таким образом. Аннотация же предоставляет эту информацию.

Располагая этой дополнительной информацией, PREfast может проверить реализацию функции `memset`, чтобы удостовериться в том, что она никогда не обращается за пределы буфера, т. е. никогда не обращается больше чем к `s` байтам буфера. PREfast также может проверить, что при вызове функции значение `p + s` всегда находится в объявленных пределах массива. В этом случае, размер выражается в байтах, т. к. функция `memset` ожидает этот формат.

PREfast поддерживает два вида аннотаций. Аннотации общего назначения определены в файле `%wdk%\inc\api\Specstrings.h`, и их можно применять как с драйверами, так и с общим кодом режима ядра и пользовательского режима. Специфичные для драйверов аннотации определены в файле `%wdk%\inc\ddk\Driverspecs.h` и предназначены специально для применения с драйверами режима ядра. Остаток этой главы посвящен подробному рассмотрению этих аннотаций.

## Использование аннотаций

Внедрение аннотаций в исходный код значительно повышает возможности PREfast по выявлению возможных ошибок и в то же самое время понижает уровень ложноположительных и ложноотрицательных результатов. Например, если добавить в код аннотацию, указывающую, что параметр представляет буфер определенного размера, то PREfast может проверять операции, которые могут вызвать переполнение буфера. Аннотации можно применять ко всей функции, к индивидуальным параметрам функции и к объявлениям `typedef`.

Аннотации не мешают нормальной компиляции на любом компиляторе, т. к. в системе аннотаций для PREfast используются макросы. Когда исполняется PREfast, то эти макросы развертываются в смысловые определения. При обычной же компиляции кода эти макросы не развертываются, не модифицируя программу никаким образом. Аннотации видимы только инструментам статического анализа, таким как PREfast, и людям, которые часто находят их очень информативными.

## Как аннотации улучшают результаты PREfast

Аннотации могут предоставить PREfast информацию о глобальном состоянии и работе, выполняемой за пределами функции, а также специфическую информацию о ролях и возможных значениях параметров функций. Эта информация позволяет PREfast анализировать код более точно, со значительно более низким уровнем ложноположительных и ложноотрицательных результатов.

## Аннотации расширяют прототипы функций

Прототипы функций предотвращают многие ошибки, устанавливая тип и число параметров функций, чтобы неправильные вызовы можно было обнаружить во время компиляции. Но прототипы не предоставляют достаточно информации о предполагаемом использовании функции, чтобы PREfast мог определить или устраниить возможные ошибки.

Например, С передает параметры по значению, поэтому сами параметры всегда являются входными параметрами функции. Но в С по значению можно также передавать указатели, поэтому из прототипа функции невозможно понять назначение передаваемого значения: является ли оно вводом в функцию, выводом из функции или и то, и другое. PREfast не может определить, требуется ли инициализировать параметр перед вызовом, и поэтому он может только пометить неинициализированный параметр, как возможную проблему. Применение аннотаций помогает уточнить назначение параметров функций.

## Аннотации описывают контракт между вызываемой функцией и вызывающим ее клиентом

Аннотации подобны статьям контракта. Как в любом контракте, обе стороны имеют обязанности и ожидают определенные результаты. Для данного контракта:

- ◆ вызывающая функция (вызывающая сторона) выполняет свои обязанности, предоставляя требуемые входные данные должным образом;
- ◆ вызываемая функция (вызываемая сторона) выполняет свои обязанности, возвращая ожидаемые результаты должным образом.

Анализируя контракт вызова функции, PREfast проверяет, чтобы вызывающая сторона выполнила свои обязательства, а для своих последующих анализов предполагает, что вызываемая сторона выполнила свою часть обязанностей должным образом.

При проверке контракта тела функции PREfast поступает наоборот: предполагает, что вызывающая сторона выполнила свои обязанности должным образом, и проверяет, чтобы вызываемая сторона выполнила свою часть обязанностей. В той мере, насколько точно аннотации представляют обязанности вызывающей и вызываемой сторон, PREfast может проверить многие взаимоотношения между функциями, которые поначалу могли бы казаться не под силу инструменту, анализирующему по одной функции за раз.

## Аннотации помогают усовершенствовать конструкцию функции

В случае хорошо сконструированной функции, вставка аннотаций в исходный код является прямолинейной задачей. С другой стороны, в случае дефектной или неполноценной конструкции функции, сам процесс внедрения аннотаций может помочь выявить важные проблемы на ранних стадиях разработки. Рассмотрим несколько примеров из этой области.

- ◆ Функция не предоставляет достаточно информации, чтобы предотвратить переполнения буфера, что указывает на возможную ошибку, которая должна быть исправлена даже до применения PREfast.
- ◆ Вставка аннотации поднимает вопрос по конструкции функции, который должен быть разрешен.

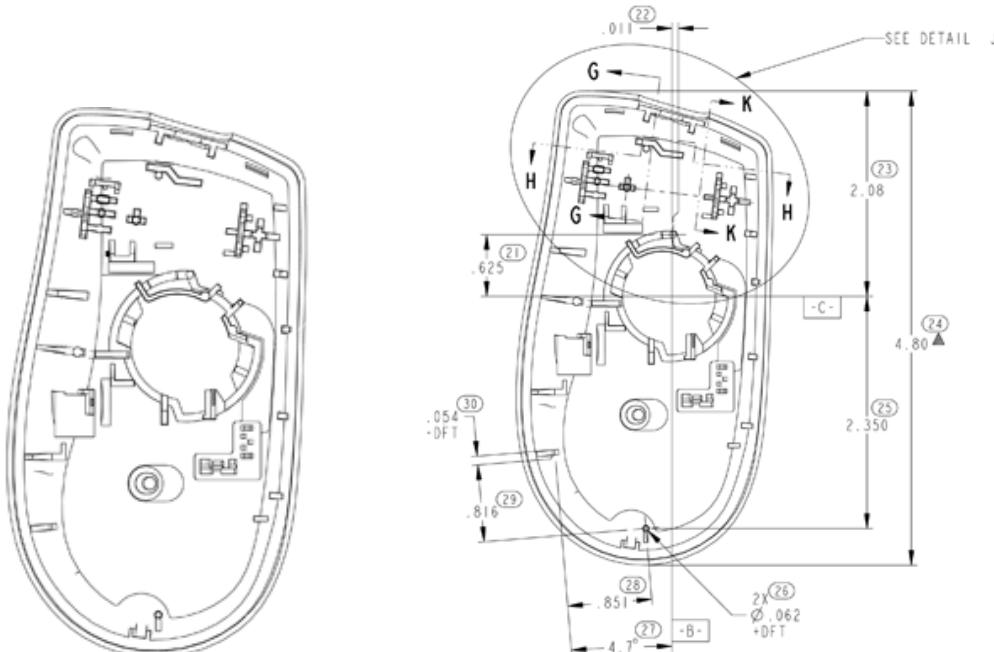
Типичным примером такой проблемы может быть вопрос, действительно ли параметр является необязательным (т. е. может ли его значение быть `NULL`).

- ◆ Для функции трудно подобрать аннотации, что может указывать на изъяны в конструкции функции.

Таким образом, даже если в ближайшее время изменить функцию невозможно, попытки аннотировать ее могут определить проблемы, которые нужно будет устранить.

### **Аннотации похожи на выноски в чертежах**

Аннотации в исходном коде можно сравнить с выносками в чертеже какой-либо детали или устройства, например нижней части корпуса мыши, показанной на рис. 23.10.



**Рис. 23.10.** Чертеж нижней части корпуса мыши

Точно так же, как и должным образом выполненный чертеж детали указывает все размеры и допуски, чтобы обеспечить производство взаимозаменяемой детали, так и четкая и поддающаяся проверке спецификация поведения функции позволяет многократное использование функции с высокой степенью уверенности в правильности ее работы. Это дает двухкратную выгоду: помогает убедиться в правильности кода, использующего функцию, и помогает с многократным использованием функции, уменьшая объем почти идентичного кода, который приходится создавать по причине незавершенности спецификации функции.

Аннотации для PREfast могут, в некоторой степени, визуально загромоздить исходный код, но без этих аннотаций картина, представляемая исходным кодом, не является полной, точно так же, как и аннотации в чертеже детали мыши уменьшают художественную чистоту чертежа, но являются необходимыми для успешного изготовления данной детали. (Дан Тэри (Donn Terry) член команды разработчиков PREfast for Drivers, Microsoft.)

### **Куда вставлять аннотации в коде**

Обычно аннотации применяются с функциями и их параметрами. Их также можно применять с объявлениями `typedef`, включая объявления типов функций.

## Аннотирование функций и параметров функций

Обычно аннотации, которые относятся ко всей функции, следует помещать непосредственно перед началом определения функции. Аннотации, которые относятся к параметру функции, можно помещать либо в строке параметра, либо заключенные в аннотации `_drv_arg` перед началом функции.

В листинге 23.4 приведен пример помещения различных аннотаций для кода общего назначения и для кода драйверов. Более подробное объяснение этих аннотаций будет приведено далее в этой главе. Целью этого примера является только показать, где именно в коде можно помещать аннотации, а не что они делают.

### Примечание

В этом и во многих других примерах в оставшейся части этой главы рассматриваемые аннотации в исходном коде выделены жирным шрифтом, чтобы отличить их от обычного исходного кода.

#### Листинг 23.4. Помещение аннотаций PREfast для функции

```
_checkReturn
_drv_allocatesMem(Pool)
_drv_when(PoolType&0x1f==2 || PoolType&0x1f==6,
_drv_reportError("Must succeed pool allocations are"
"forbidden. Allocation failures cause a system crash"))
// Выделения из пулов, которые должны быть успешными, запрещены.
// Неуспешные операции выделения могут вызвать системный сбой.
_bcount(NumberOfBytes)
VOID ExAllocatePoolWithTag(
    _in _drv_strictTypeMatch(drv_typeExpr) POOL_TYPE PoolType,
    _in SIZE_T NumberOfBytes,
    _in ULONG Tag
);
```

Обсудим данный пример.

- ◆ Аннотации `_checkReturn`, `_drv_allocatesMem`, `_drv_when` и `_drv_reportError` относятся к функции `ExAllocatePoolWithTag`:
  - аннотация `_checkReturn` дает указание PREfast выдать предупреждение, если последующий код игнорирует возвращаемое функцией значение;
  - аннотация `_drv_allocatesMem` указывает, что функция выделяет память, в данном случае — страницу;
  - аннотация `_drv_when` указывает условное выражение: если функция вызывается с одним из типов пула "must succeed" (операция должна быть успешной), PREfast должен выводить сообщение об ошибке, указанное в аннотации `_drv_reportError`.
- ◆ Аннотация `_bcount` относится к возвращаемому функцией значению `VOID`. Эта аннотация указывает, что возвращаемое значение должно иметь `NumberOfBytes` количество байтов.
- ◆ Аннотация `_in _drv_strictTypeMatch` относится к параметру `PoolType`. Эта аннотация указывает, что параметр может принимать только типы, подразумеваемые аннотацией `_drv_typeExpr`, которые могут быть либо константы, либо выражения, содержащие операнды только определенного типа.

- ◆ Остальные аннотации типа `_in` относятся к параметрам `NumberOfBytes` и `Tag` соответственно. Эти аннотации указывают, что PREfast должен проверить, что при входе в функцию эти параметры являются действительными.

Обычно при употреблении с параметрами простые аннотации, такие как `_in`, являются более удобочитаемыми, а вставка в тело функции более сложных аннотаций может сделать код трудночитаемым. Вместо помещения аннотаций для параметров в тело функции их можно заключить в аннотации `_drv_arg` и поместить вместе с другими аннотациями перед началом функции, что будет способствовать улучшению читаемости более сложных аннотаций.

Например, в листинге 23.5 показан исходный код функции `ExAllocatePoolWithTag`, в которой аннотации для параметра `PoolType` заключены в аннотацию `_drv_arg` и помещены в начале функции, а не в теле функции.

**Листинг 23.5. Альтернативный вариант помещения аннотаций PREfast для функции с параметрами**

```
_checkReturn
_drv_allocatesMem(Pool)
_drv_when(PoolType&0x1f==2 || PoolType&0x1f==6,
    _drv_reportError("Must succeed pool allocations are"
    "forbidden. Allocation failures cause a system crash"))
// Выделения из пулов, которые должны быть успешными, запрещены.
// Неуспешные операции выделения могут вызвать системный сбой.
_drv_arg(PoolType, _in _drv_strictTypeMatch(_drv_typeExpr))
_bcount(NumberOfBytes)
PVOID
ExAllocatePoolWithTag(
    POOL_TYPE PoolType,
    _in SIZE_T NumberOfBytes,
    _in ULONG Tag
);
```

Более подробная информация об аннотациях `_checkReturn`, `_bcount` и `_in` приводится в разд. "Аннотации общего назначения" далее в этой главе.

Дополнительная информация об аннотациях `_drv_arg`, `_drv_allocatesMem`, `_drv_when`, `drv_reportError` и `_drv_strictTypeMatch` приводится в разд. "Аннотации для драйверов" далее в этой главе.

### Аннотации для объявлений `typedef`

Аннотации, применяемые с объявлениями `typedef`, неявно накладываются на функции и параметры данного типа. Поэтому, при наложении аннотаций на объявление `typedef`, включая объявления `typedef` функций, накладывать аннотации на применения данного типа не требуется. PREfast интерпретирует аннотации на объявление `typedef` таким же образом, как он интерпретирует аннотации для функций.

Использование аннотаций с объявлениями `typedef` является как более удобным, так и более безопасным, чем применение аннотаций с каждым отдельным параметром функции данного типа. Возьмем, например, функцию, которая принимает строку с завершающим нулем в качестве параметра. В языке С между массивом символов и строкой нет никакой разницы, кроме того, что строка завершается нулем. Но семантика многих функций полагается на

знание или предположение, что определенный массив символов оканчивается нулевым символом и поэтому является строкой. Если PREfast знает, что массив типа `char` или `wchar_t` должен быть строкой, то он может выполнить дополнительные проверки массива, чтобы удостовериться в том, что он должным образом заканчивается нулевым символом. Эту информацию можно выразить элементарной аннотацией `_nullterminated`.

В принципе, аннотацию `_nullterminated` можно было бы явно применять с каждым параметром функции, принимающим строку в качестве параметра, как показано в следующем примере:

```
size_t strlen(_nullterminated const char *s);
```

Но процесс накладывания аннотаций таким образом быстро становится утомительным и предрасполагает к ошибкам. Вместо этого можно объявить строковый параметр как тип, к которому уже применена аннотация `_nullterminated`. В таком случае не требуется явно аннотировать все функции, которые используют строки, чтобы удостовериться в том, что их параметры заканчиваются завершающим нулем. Примером такой аннотации может служить следующее объявление `typedef` для `PCSTR`:

```
typedef _nullterminated const char *LPCSTR, *PCSTR;
```

С применением такой аннотации объявление функции становится намного проще, как показано в следующем примере:

```
size_t strlen(PCSTR s);
```

В данном примере выражение `PCSTR s` подразумевает, что `s` заканчивается завершающим нулем, т. к. для типа `PCSTR` применена аннотация `_nullterminated`. Такое объявление более удобочитаемое, чем объявление в предыдущем примере, и выражает предполагаемое использование параметра более ясно.

В определяемых функциях строки всегда должны быть типа `PCSTR` или подобного типа. Тип `PCHAR` должен применяться только для строк, которые не заканчиваются завершающим нулем. Применение типа `PCSTR` и подобных типов для строк предоставляет преимущества аннотаций без необходимости явного их помещения, а отличить функцию, принимающую в качестве параметра строку, от функции, принимающей массив байтов в виде 8-битных чисел, не составляет труда. Но если тип `PCSTR` или подобные типы применяются для описания массива байтов, который может не заканчиваться завершающим нулем, то применение неявной аннотации `_nullterminated` на строковом типе причиняет PREfast выдавать ложноположительные результаты.

Разница между строкой и массивом байтов демонстрируется в листинге 23.6. Функция `FindChar` полагается на неявное гарантирование типом параметра `PSTR`, что строка заканчивается завершающим нулем. Самостоятельно функция `FindChar` не может обнаружить завершающий ноль в теле строки.

Более реалистичным примером было бы использование таких аннотаций, как `_drv_when` и `_drv_reportError`, чтобы указать, что `s` должно быть ненулевым символом. Эти аннотации рассматриваются более подробно дальше в этой главе.

Функция `FindByte` полагается на неявное гарантирование типом параметра `PCHAR` того, что ноль не является специальным символом в массиве `arr`, и того, что переменная `len` определяет размер массива, в котором следует выполнить поиск, поэтому для целей поиска двоичный ноль является допустимым символом. Когда PREfast анализирует функцию `FindChar`, то выполняет проверку на отсутствие требуемого символа конца строки в переменной `str`, т. к.

типа параметра `PSTR` указывает, что буфер по адресу памяти `str` должен быть строкой, а не массивом.

#### Листинг 23.6. Разница между строкой и массивом байтов

```
PSTR FindChar( _in PSTR str, _in char c)
{
    // Переменная str заканчивается символом '\0',
    // поэтому нужно прекратить поиск, когда увидим 0.
    // Длина строки не требуется.
    // Переменная с не может быть 0, т. к. в таком случае
    // совпадения никогда не будет.

    }

PCHAR * FindByte(_in PCHAR *arr, long len, _in char b)
{
    // Мы не знаем, является ли последний символ в буфере нулем
    // или нет: мы просто полагаемся на длину буфера и прекращаем
    // поиск после просмотра len числа байтов.
    // Значение переменной b может быть 0: в данном случае ноль –
    // такой же символ, как и все другие.

}
```

### Аннотации к объявлениям `typedef` функций

В языках С и C++ `typedef` можно применять для объявления типа функции. Это употребление отличается от типа `pointer` функции и традиционно широкого применения в программах на языке С не имело. Но с введением аннотаций и классов типов функций, которые описываются далее в этой главе, типы функций становятся очень полезными.

Хотя объявления функций с применением `typedef` могут казаться необычными, они являются действительным стандартом языка С, т. е. они правильны и поддаются компиляции. Например, в листинге 23.7 показан пример использования `typedef` для объявления функции `DRIVER_STARTIO`, которая определена в файле `%wdk%\inc\ddk\Wdm.h`.

#### Листинг 23.7. Применение `typedef` для объявления функции `DRIVER_STARTIO`

```
typedef
VOID
DRIVER_STARTIO (
    _in struct _DEVICE_OBJECT *DeviceObject,
    _in struct _IRP *Irp
);
typedef DRIVER_STARTIO *PDRIVER_STARTIO;
```

Важно помнить, что `DRIVER_STARTIO`, определяет `typedef` функции, а не `typedef` указателя функции. Это объявление функции `typedef` используется, чтобы объявить, что функция `MyStartIo` является функцией типа `DRIVER_STARTIO`.

Функция, объявленная с `DRIVER_STARTIO`, является совместимой с точки зрения присваивания со знакомым указателем функции `PDRIVER_STARTIO`, т. е. указатель на одну из этих функций можно присвоить указателю на другую.

Также обратите внимание на то, что параметры функции в `DRIVER_STARTIO` уже имеют аннотации `_in`, что идентифицирует их как входные параметры. Эти аннотации делаются неявно в объявлении `typedef`, поэтому аннотировать эти параметры в функции нет надобности, разве что для улучшения читаемости.

Если функция `MyStartIo` должна быть функцией `StartIo` WDM, ее можно объявить как функцию типа `DRIVER_STARTIO`, поместив следующее объявление перед первым использованием функции `MyStartIo` в драйвере:

```
DRIVER_STARTIO MyStartIo;
```

Кроме объявления `MyStartIo` функцией типа `DRIVER_STARTIO`, это объявление применяется к `MyStartIo` все системные аннотации на типе `DRIVER_STARTIO`, как определено в файле `%wdk%\inc\Wdm.h`.

После объявления `typedef` функции не обязательно давать полный прототип функции. Многие разработчики находят индивидуальное объявление функции `typedef` более читаемым. Тело функции `MyStartIo` (т. е. ее определение) реализуется таким же образом, как и любой другой функции. Для примера драйвера, использующего объявления функции `typedef` и опускающего прототипы, см. образец драйвера `Toaster` в наборе разработчика WDK в папке `%wdk%\src\general\toaster\func\shared\toaster.h`.

Объявление функции `typedef` имеет еще одно преимущество: оно дает знать другим программистам, что функция должна быть настоящей функцией `StartIo`, а не просто выглядеть, как такая функция.

Например, функции `Cancel` являются совместимыми с точки зрения присваивания с функциями `StartIo`, поэтому плохо продуманное имя функции может вызвать неопределенности в исходном коде.

С помощью аннотации `_drv_functionClass` PREfast можно также указать, что тип функции относится к определенному классу типа функции. Это значительно увеличивает объем проверок, которые может выполнить PREfast, т. к. PREfast понимает, что эта функция является функцией обратного вызова, и знает, что должны соблюдаться условия конкретного контракта. Этот вопрос рассматривается более подробно в разд. "Аннотации класса типа функций" далее в этой главе.

При использовании объявлений функций `typedef` помните следующее.

- ◆ Тип функции должен строго соответствовать типу, объявляемому оператором `typedef` для функции.
- ◆ Объявление функции `typedef` должно иметь необходимые аннотации. Системные объявления `typedef` уже имеют эти аннотации.
- ◆ Объявление должно предшествовать первому упоминанию функции. Если функция фигурирует в заголовочном файле, новое объявление должно предшествовать упоминанию функции в заголовке или же просто заменить его. Если первым упоминанием функции является само определение функции, то тогда объявление должно непосредственно предшествовать этому объявлению.
- ◆ Определение функции аннотировать не требуется, т. к. аннотации, наложенные на объявления функции `typedef`, неявно накладываются на функции и параметры данного типа, как и для любого объявления `typedef`.
- ◆ Если аннотировать каждый параметр отдельно, то объявления функций `typedef` могут стать трудночитаемыми. Чтобы сделать объявления функций `typedef` удобочитаемыми,

можно аннотировать объявление `typedef` для каждого параметра, после чего объявить каждый параметр соответствующего типа в объявлении функции `typedef`.

### Примечание

Объявления функций `typedef` в драйверах, такие как `DRIVER_STARTIO`, предназначены для использования в драйверах, написанных на языке С. Самые последние доклады, ресурсы и полезные советы по применению PREfast для тестирования см. на странице по PREfast на Web-сайте WHDC.

## Советы по размещению аннотаций в исходном коде

Далее приводится несколько полезных советов по размещению аннотаций в исходном коде.

- ◆ Если для функции существует как объявление (т. е. прототип), так и определение, для обоих компонентов необходимо применять идентичные аннотации. Объявления функций `typedef` помогают удовлетворить это требование.  
В случае несовпадения объявлений, PREfast выдает предупреждение.
- ◆ Если функция имеет только определение и не имеет отдельного прототипа, аннотируйте определение. Если нужно только вставить аннотации, и прототип не нужен ни для чего другого, то создавать и аннотировать его нет необходимости.
- ◆ Аннотации, которые относятся ко всей функции, следует помещать непосредственно перед началом функции.
- ◆ Аннотации, которые относятся к параметру функции, можно помещать или в строку параметра непосредственно перед параметром, или непосредственно перед началом функции, заключенной в аннотацию `_drv_arg`, идентифицирующую параметр, к которому относится данная аннотация.
- ◆ Аннотируйте объявления `typedef`, чтобы неявно помещать их в функции и параметры данного типа.

## Аннотации общего назначения

Аннотации общего назначения предоставляют PREfast сведения о потоке информации между клиентом и вызываемой функцией, относительно как направления потока информации, так и размера и типа информации, которую можно проверить, чтобы выявить возможные переполнения буфера.

Аннотации общего назначения можно применять как с кодом драйверов, так и с обычным кодом. Эти аннотации определены в файле `Specstrings.h` и подробно описаны в файле `Specstrings_strict.h`. Оба файла находятся в папке `%wdk%\inc\api`.

В этом разделе предоставляются рекомендации и примеры по использованию аннотаций общего назначения и модификаторов, перечисленных в табл. 23.1.

**Таблица 23.1. Аннотации общего назначения**

| Аннотация                      | Применение                        |
|--------------------------------|-----------------------------------|
| <code>_in, _out, _inout</code> | Для входных и выходных параметров |
| <code>_opt, _deref</code>      | Модификаторы аннотаций            |

Таблица 23.1 (окончание)

| Аннотация                                                                      | Применение                     |
|--------------------------------------------------------------------------------|--------------------------------|
| <code>_ecount(size), _bcount(size)</code>                                      | Размер параметра               |
| <code>_full(size), _part(size, length)</code>                                  | Частичный размер параметра     |
| <code>_nullterminated, _nullnullterminated, _possibly_notnullterminated</code> | Строковые аннотации            |
| <code>_reserved</code>                                                         | Зарезервированные параметры    |
| <code>_checkReturn</code>                                                      | Возвращаемое функцией значение |

PREfast не рассматривает определенные аннотации, такие как, например, `_fallthrough`, но и эти аннотации могут быть полезными, если помещены в код в виде комментариев. Полный список аннотаций см. в комментарии в файле %wdk%\inc\api\Specstrings\_strict.h.

Аннотации могут быть элементарными и составными. Составная аннотация состоит из двух или более элементарных аннотаций и других составных аннотаций. В этой главе объясняются некоторые аннотации на основе элементарных аннотаций. Но во всех возможных случаях следует применять составные аннотации вместо примитивных, т. к. составные аннотации являются более устойчивыми к будущим изменениям.

### Внимание!

Для простоты, примеры в этой главе показывают стандартные функции, которые не полностью аннотированы. Но для своего кода вам следует применять все соответствующие аннотации, как описано в этой главе, чтобы обеспечить своим функциям полную аннотированность и полный анализ со стороны PREfast. Не ограничивайтесь в своем исходном коде только аннотациями, показанными в примерах данной главы.

## Аннотирование входных и выходных параметров

В языке C параметры передаются по значению, поэтому сами параметры всегда являются входными параметрами функции. Но т. к. в C по значению можно также передать указатели, то только из прототипа функции невозможно понять назначение передаваемого с помощью указателя значения: является ли оно вводом для функции, выводом для функции или и тем, и другим.

Аннотации `_in`, `_out` и `_inout` позволяют PREfast проверять параметры с этими аннотациями и докладывать обо всех ошибках, если он обнаруживает, что неинициализированные значения используются неправильно.

- ◆ Если параметр имеет аннотацию `_in`, то PREfast выполняет проверку, чтобы убедиться в том, что параметр инициализируется перед вызовом.

Аннотацию `_in` можно также применять со скалярными параметрами, такими как целые числа или перечислимые типы. Для скалярных параметров аннотация `_in` не является обязательной, но ее применение помогает сделать код более единообразным и читаемым.

- ◆ Если параметр обозначен аннотацией `_out`, то PREfast не проверяет его инициализацию перед вызовом, но предполагает, что он инициализируется после вызова и поэтому является безопасным для использования в качестве параметра `_in` для следующей функции.

- ◆ Если параметр обозначен аннотацией `_inout`, то PREfast проверяет его инициализацию перед вызовом и предполагает, что он является безопасным для использования в качестве параметра `_in` для следующей функции.

### Полезная информация

Для непрозрачных типов обычно достаточно лишь аннотации `_in`, включая дескрипторы KMDF, такие как `WDFDEVICE`.

### Контракт аннотаций `_in` и `_out`

Термин "инициализированный" применяется неформально при обсуждении аннотаций `_in` и `_out`, но эти аннотации в действительности описывают то, что делает аннотированные параметры действительными по отношению к предшествующему и последующему состоянию.

- ◆ "Действительный" означает, что все уровни структуры данных инициализированы и что указатели на всех уровнях разыменования за исключением последнего указателя являются ненулевыми значениями, кроме случаев, когда к параметру была явно применена другая аннотация.
- ◆ Предшествующее состояние (pre-state) означает состояние анализа непосредственно перед вызовом функции. Если переменной было присвоено значение в предыдущем присваивании, тогда она имеет это значение в предшествующем состоянии.
- ◆ Последующее состояние (post-state) означает состояние сразу же после возвращения управления вызванной функцией. Для параметров `_out` последующее состояние — это состояние, в котором параметр имеет новое значение или значение вообще.

Формально, аннотация `_in` означает, что передаваемое в качестве аргумента значение должно быть действительным в предшествующем состоянии, и функция не изменяет это значение. Аннотация `_out` означает, что функция заключила контракт на возврат действительного значения в последующем состоянии и что PREfast может игнорировать предшествующее состояние.

В предшествующем состоянии аннотация `_out` подразумевает, что любые промежуточные указатели, ведущие к конечному значению, подлежащему модифицированию, являются индивидуально действительными и не нулевыми, но параметр, обозначенный аннотацией `_out`, не обязан быть рекурсивно действительным в предшествующем состоянии. Например, допустим, что необходимо передать указатель на указатель на структуру `s` (т. е. нужно передать `**p`). В предшествующем состоянии для этого необходимо, чтобы как `p`, так и `*p` были действительными указателями на другие указатели. Но для параметра `_out` указатель `**p` не обязан быть действительным, т. е. в предшествующем состоянии указатель `**p` не обязан быть полностью инициализированным.

Для аннотации `_out` в предшествующем состоянии структура `**s` должна быть действительной (т. е. ожидается, что структура будет заполненной). Модификатор `_opt` можно использовать, чтобы указать, можно ли опустить какие-либо промежуточные уровни, например, может ли `*p` иметь значение `NULL`.

Аннотация `_inout` означает, что параметр должен быть действительным, как в предшествующем, так и в последующем состоянии, и что оба состояния должны быть проверены. Она также означает, что любые предположения PREfast о значении параметра, за исключением его действительности, в последующем состоянии больше не являются достоверными. То есть, предполагается, что значение изменилось.

Как было сказано ранее в этой главе, PREfast анализирует обе стороны контракта. Аннотации `_in`, `_out` и `_inout` ясно иллюстрируют это утверждение. Для анализируемой функции (т. е. для вызываемой стороны) PREfast предполагает, что предшествующее состояние является достоверным в точке входа, и проверяет, чтобы в точке выхода было достигнуто последующее состояние. В частности, анализ функции начинается с первоначального состояния параметров. PREfast предполагает, что при вызове функции все параметры `_in` являются действительными, и проверяет действительность всех параметров `_out` при возвращении управления функцией.

В следующих двух примерах показано, как PREfast анализирует контракты согласно аннотациям. Аннотация `_in` в следующем объявлении функции `strlen` заставляет PREfast проверить, что перед вызовом функции входной параметр `s` инициализирован:

```
size_t strlen(_in PCSTR s);
```

А аннотация `_nullterminated`, неявно присутствующая в таких строковых типах, как `PSTR`, `PWSTR` и т. п., проверяет обе стороны контракта, как показано в следующем примере:

```
PSTR substitute(_inout_ecount(len) PSTR str,
                _in int len, _in PSTR oldstr, _in PSTR newstr);
```

В данном примере функция `substitute` принимает в качестве входного параметра строку и заменяет в ней все случаи значения переменной `oldstr` значением переменной `newstr`. Функция никогда не выходит за пределы `len` байт, и обеспечивает, что полученное значение `str` всегда оканчивается завершающим нулем. Допуская, что функция замены выполняет какую-либо работу, часть `_inout` аннотации, применяемая к переменной `str`, указывает, что перед вызовом функции и после возвращения из нее эта переменная имеет разные значения, но что она является действительной как до вызова функции, так и после возвращения из нее. Относительно данного контракта PREfast выполняет следующие анализы.

- ◆ Для вызывающей части контракта PREfast проверяет, чтобы, насколько это можно определить статическим образом, переменные `str`, `oldstr` и `newstr` оканчивались завершающим нулем в точке вызова. Также проверяется, чтобы буфер по адресу `str` мог вместить `len` байт.
- ◆ Для вызываемой части контракта PREfast проверяет, чтобы обращение к `str` не выходило за пределы `len` байт. PREfast предполагает, что значения переменных `str`, `oldstr` и `newstr` оканчиваются завершающим нулем. Критической проверкой, выполняемой PREfast, является проверка, чтобы конечное значение переменной `str` оканчивалось завершающим нулем.

Большинство аннотаций, описываемых далее в этой главе, включая аннотации для уровней IRQL, памяти и ресурсов иных, чем память, имеют семантику, как для вызывающей, так и для вызываемой стороны, обеспечивающую выполнение контракта обеими сторонами.

### **Аннотации `_in`, `_out` и `_inout` и макросы `IN`, `OUT` и `INOUT`**

В общем, все случаи макросов `IN`, `OUT` и `INOUT` следует заменять аннотациями `_in`, `_out` и `_inout` соответственно. Но это не означает, что эти устаревшие макросы можно просто переопределить на основе аннотаций.

Хотя макросы `IN` и `OUT` часто встречаются в исходном коде, им никогда не присваивается значение, и они никогда не проверяются никаким инструментом или компилятором; поэтому они не всегда отражают действительное использование параметров и могут быть неправильными. Поэтому эти макросы могут быть неправильно размещены или использоваться в

исходном коде. Следует проанализировать функции, использующие макросы `IN`, `OUT` и `INOUT`, и обязательно вставить аннотации `_in`, `_out` и `_inout` в соответствующие места в коде.

## Модификаторы аннотаций

По различным причинам, связанным с реализацией, многие аннотации, которые нужно применить к параметрам функции, необходимо представлять в виде одного макроса, а не последовательности смежных макросов. В особенности это относится к большинству различных базовых аннотаций, которые для каждого параметра должны быть в виде одного составного макроса.

Это требование выполняется добавлением к аннотации модификатора, чтобы составить более полную аннотацию. В качестве примеров создания более сложных аннотаций из более простых аннотаций рассмотрим два наиболее употребляемых модификатора — `_opt` и `_deref`.

### Модификатор `_opt`

Аннотация `_in` не допускает нулевых указателей, но функции часто могут принимать `NULL` вместо настоящего параметра. Модификатор `_opt` указывает, что параметр является необязательным, т. е. что он может быть `NULL`. Например, для необязательного входного параметра (например, указателя на структуру) аннотация была бы `_in_opt`, а для необязательного выходного параметра — `_out_opt`.

Обычно аннотации `_in_opt` и `_out_opt` применяются для указателей на структуры неизменяемого размера. Дополнительные модификаторы можно использовать для аннотирования объектов изменяемого размера, как описано в разд. "Аннотации размера буфера" далее в этой главе.

В общем, следует заменять все случаи макроса `OPTIONAL` модификатором `_opt`. Но т. к. достоверность макроса `OPTIONAL` не проверяется никаким инструментом или компилятором, необходимо внимательно проанализировать код, чтобы удостовериться в том, что параметры, помеченные `OPTIONAL`, на самом деле являются необязательными.

### Модификатор `_deref`

Определяемые пользователем типы, такие как, например, структуры, можно объявлять как типы параметров, поэтому иногда необходимо проаннотировать разыменованное значение параметра. Модификатор `_deref` указывает, что аннотацию следует применять к разыменованному значению параметра, а не к самому параметру.

Например, рассмотрим следующую функцию:

```
int myFunction(struct s **p);
```

Когда функции передается такой указатель, как `struct s *p`, то область памяти, на которую указывает `*p`, передается по ссылке, а сама переменная `p` передается по значению. В данном примере, параметр `p` является переменной типа "указатель на `s`", которая передается по ссылке. `**p` является переменной типа `struct s`, `*p` — указатель на эту переменную, а `p` — указатель на этот указатель.

В данном примере функция `myFunction` определена, чтобы модифицировать указатель `*p` на переменную типа `struct s`. Значение `p` для функции должно быть ненулевым. Но функция разрешает значение `NULL` для `*p` — в таком случае функция просто ничего не делает с `*p`.

Применив к переменной `p` аннотацию `_inout`, можно было бы создать требование, чтобы `*p` не было `NULL`. А применив к переменной `p` аннотацию `_inout_opt`, можно было позволить значению `p` быть `NULL`. Но ни одна из этих аннотаций не выражает должным образом желаемое поведение функции `myFunction`.

Добавление `_deref` модификатора к аннотации `_inout_opt` относит ее к правильному разыменованному значению `p`, как показано в следующем примере:

```
int myFunction(_inout_deref_opt struct s **p);
```

Данная составная аннотация указывает, что элементарная аннотация `_opt` относится к `*p`, которое является разыменованным значением переменной `p`, т. е. `*p` может быть `NULL`. Аннотация `_opt` не относится к самой `p`, т. е. `p` не может быть `NULL`. Иными словами, аннотация `_deref_opt` распространяется на параметр, передаваемый по ссылке (т. е. на `*p`), а не адрес ссылки — `p`.

Модификатор `_deref` можно применять в аннотации несколько раз, чтобы указывать несколько уровней разыменования. Например, аннотация `_in_deref_deref_opt` означает, что `**p` может быть `NULL`. Применение модификатора `_deref` с другими аннотациями показано во многих примерах далее в этой главе.

### Примечание

Аннотации `_null` и `_notnull`, которые явно указывают, что определенный параметр может быть `NULL` или должен быть не `NULL`, встроены в составные аннотации общего назначения, такие как `_inout`. Вставлять аннотации `_null` и `_notnull` в аннотации, подобные показанным в этом примере, не требуется.

## Аннотации размера буфера

Объектом изменяемого размера является любой объект, который не несет с собой информацию о своем размере. Многие ошибки в коде, особенно ошибки безопасности, вызываются переполнениями буфера при передаче в буфер объекта изменяемого размера. Контракт между вызывающим и вызываемым компонентами относительно размера буферов можно выразить с помощью следующих аннотаций:

```
_ecount(size)
_bcount(size)
_full(size)
_part(size, length)
```

Контракт должен указать размер или место, где можно взять эту информацию, и его использование. Эта информация обеспечивает, что ни вызывающий, ни вызываемый компоненты не будут обращаться к данным за пределами буфера; кроме этого, контракт также может выражать разницу между доступной и инициализированной памятью, с тем, чтобы можно было обнаруживать доступ к неинициализированной памяти.

В языке С буфера обычно являются массивами данных определенного типа. В коде размер буфера можно выразить двумя способами: в виде числа байтов в буфере или в виде количества элементов буфера. Для массивов любого другого типа, кроме `char`, размер в элементах отличается от размера в байтах. В большинстве случаев, даже для массивов типа `char`, размер в элементах более полезен, и его легче выразить.

## Примечание

Размер широких символьных строк, таких как `wchar_t`, обычно выражается в элементах, а не байтах. Исключением этого являются строки типа `UNICODE_STRING`.

## Аннотации буфера неизменяемого размера

Аннотации `_ecount(size)` и `_bcount(size)` применяются для выражения размера буфера. Аннотация `_ecount(size)` применяется для указания размера буфера в виде количества элементов. А аннотация `_bcount(size)` служит для указания размера буфера в виде количества байтов. Параметр `size` может быть любым общим выражением, имеющим смысл во время компиляции. Это может быть число, но обычно это имя какого-либо параметра в функции, подвергающейся аннотации.

В следующем примере функции `memset` показана типичная аннотация буфера:

```
void * memset(
    _out_bcount(s) char *p,
    _in int v,
    _in size_t s);
```

В данном примере аннотация `_out_bcount(s)` указывает, что содержимое области памяти, указываемой параметром `p`, устанавливается функцией, и что количество байтов, которые нужно установить, указывается в `s`. В исходном коде С нет ничего, что могло бы сказать компилятору, что переменные `p` и `s` связаны таким образом. Аннотация же предоставляет эту информацию.

Располагая этой дополнительной информацией, PREfast может проверить реализацию функции `memset`, чтобы удостовериться в том, что она никогда не обращается за пределы буфера, т. е. никогда не обращается больше чем к `s` байтам буфера. Часто PREfast также может проверить, что при вызове функции значение `p + s` всегда находится в объявленных пределах массива. В этом случае размер выражается в байтах, т. к. функция `memset` ожидает этот формат.

Сравните функцию `memset` с похожей функцией `wmemset`:

```
wchar_t * wmemset(
    _out_ecount(s) wchar_t *p,
    _in wchar_t v,
    _in size_t s);
```

В этом примере аннотация `_out_ecount` указывает, что переменная `s` представлена в элементах типа `wchar_t`. Если какой-либо посторонний код вызовет эту функцию с размером в байтах, а такую ошибку легко сделать, значение `s`, скорее всего, будет вдвое больше положенного. С помощью аннотации `_out_ecount` PREfast имеет хорошую возможность выявить ошибку переполнения буфера в вызывающем компоненте.

Обратите внимание, что для параметров `_in` определение "действительный" требует, чтобы весь передаваемый параметр был инициализирован. Это также относится и к массивам, которые передаются по ссылке.

Таким образом, при использовании аннотации `_in` для параметра, являющегося массивом, весь массив должен быть инициализирован до предела, указанного в `_bcount` или `_ecount`. Подробную информацию о том, как это относится к строкам, оканчивающимся завершающим нулем, см. в разд. "Аннотирование строк" далее в главе.

Аннотаций `_bcount` и `_ecount` достаточно, чтобы описать немодифицированные буферы типа `_in` или полностью инициализированные буферы типа `_out`. Для частично инициализированных буферов, в которых инициализированная часть буфера может быть увеличена или уменьшена по месту, эти аннотации можно совместить с модификаторами `_part` и `_full`:

- ◆ модификатор `_full` относится ко всему буферу. Для выходного буфера модификатор `_full` указывает, что функция инициализирует весь буфер. Для входного буфера модификатор `_full` указывает, что буфер уже инициализирован, хотя это повторяет другие аннотации;
- ◆ модификатор `_part` указывает, что функция инициализирует часть буфера и явно задает размер инициализированной части.

При совместном использовании этих модификаторов с буферами типа `_inout` и аннотациями `_full(size)` или `_part(size, length)` они могут представлять размеры буфера в предшествующем и последующем состоянии. Параметры `size` и `length` могут быть константами или же параметрами аннотируемой функции. В следующих примерах показано использование параметров `size` и `length` при аннотировании буферов.

- ◆ Аннотация `_inout_bcount_full(cb)` описывает буфер размером в `cb` байт, полностью инициализированный на входе и выходе, и в который данная функция, возможно, может записывать данные.
- ◆ Аннотация `_out_ecount_part(count, *countOut)` описывает буфер размером в `count` элементов, который будет частично инициализирован этой функцией. Функция указывает количество инициализированных элементов, устанавливая `*countOut`.

## Сводка аннотаций для буферов

В этом разделе дается сводка аннотаций, которые можно объединить для описания буфера. Список этих аннотаций приведен в табл. 23.2.

**Таблица 23.2. Аннотации для буферов**

| Уровень                 | Применение          | Размер                          | Вывод              | Необязательный    | Параметры                   |
|-------------------------|---------------------|---------------------------------|--------------------|-------------------|-----------------------------|
| Опущено                 | Опущено             | Опущено                         | Опущено            | Опущено           | Опущено                     |
| <code>_deref</code>     | <code>_in</code>    | <code>_ecount</code>            | <code>_full</code> | <code>_opt</code> | <code>(size)</code>         |
| <code>_deref_opt</code> | <code>_out</code>   | <code>_bcount</code>            | <code>_part</code> |                   | <code>(size, length)</code> |
|                         | <code>_inout</code> | <code>_xcount(выражение)</code> |                    |                   |                             |

Значения терминов в табл. 23.2 следующие.

**Уровень** описывает уровень разыменования указателя буфера от параметра или возвращаемого значения `p`. Уровень может быть одним из следующих:

- ◆ опущено — указателем буфера является `p`;
- ◆ `_deref` — указателем буфера является `*p`; `p` не должно быть `NULL`;
- ◆ `_deref_opt` — указателем буфера является `*p`; `p` может быть `NULL`, при этом остаток аннотации можно игнорировать.

**Применение** описывает использование буфера функцией. Применение может быть одним из следующих:

- ◆ опущено — обращение к буферу не выполняется. Если применяется для возвращаемого значения или с модификатором `_deref`, функция предоставляет буфер, и буфер инициализируется на выходе. В противном случае буфер должен быть предоставлен вызывающим компонентом. Разрешается только для функций `alloc` и `free`;
- ◆ `_in` — буфер используется только для ввода. Вызывающий компонент должен предоставить буфер и инициализировать его;
- ◆ `_out` — буфер используется только для вывода. Если применяется для возвращаемого значения или с модификатором `_deref`, функция предоставляет буфер и инициализирует его. В противном случае вызывающий компонент должен предоставить буфер, а функция инициализирует его;
- ◆ `_inout` — функция может беспрепятственно читать и записывать буфер. Вызывающий компонент должен предоставить буфер и инициализировать его. Если применяется с модификатором `_deref`, то функция может повторно выделить буфер.

**Размер** описывает полный объем буфера. Размер может быть меньшим, чем пространство в действительности выделенное буферу; в таком случае аннотация описывает доступный объем. Размер может быть одним из следующих:

- ◆ опущено — размер буфера не указан. Если размер буфера указывается типом, таким как, например, `LPSTR` или `LPWSTR`, то применяется этот размер. В противном случае размер буфера составляет один элемент. Эти аннотации применяются с `_in`, `_out` или `_inout`;
- ◆ `_ecount` — размер буфера указывается явно количеством элементов;
- ◆ `_bcount` — размер буфера указывается явно количеством байтов;
- ◆ `_xcount (выражение)` — размер буфера нельзя выразить простым указанием количества байтов или элементов. Например, количество байтов или элементов может быть в глобальной переменной, в элементе структуры или неявно указано перечислением. PREfast рассматривает выражение как примечание и не использует его для проверки размера буфера. Выражением может быть все, что понятно читателю, например настоящее выражение или строка в кавычках.

### Внимание!

Аннотация `_xcount` позволяет аннотировать буфер, но она вынуждает PREfast не выполнять проверки размера. Аннотацию `_xcount` можно использовать в качестве заполнителя для будущих работающих аннотаций, которые могут появиться в будущих инструментах. Но будьте осторожны и сдержаны с применением этой аннотации, т. к. она подавляет анализ и возможные предупреждения.

Аннотации типа **Вывод** описывают, какая часть буфера инициализируется функцией. Для буферов `_inout` эти частичные аннотации также описывают, какая часть буфера инициализируется на входе. Для буферов типа `_in` аннотации этой категории не применяются, т. к. эти буферы полностью инициализируются вызывающим компонентом.

Вывод может быть одним из следующих:

- ◆ опущено — размер инициализированной части буфера указывается типом. Например, функция, инициализирующая строку типа `LPWSTB`, должна завершить ее нулем;
- ◆ `_full` — функция инициализирует весь буфер;

- ◆ `_part` — функция инициализирует часть буфера и явно указывает размер инициализированной части.

Аннотации типа **Необязательный** указывают, является ли сам буфер обязательным. Аннотации этого типа могут быть следующими:

- ◆ опущено — указатель буфера должен быть не `NULL`;
- ◆ `_opt` — указатель буфера может быть `NULL`. Перед разыменованием PRefast проверяет указатель.

**Параметры** применяются для явного указания размера и длины буфера. Параметры `size` и `length` могут быть либо константными выражениями, либо выражениями с параметром, который обычно не является аннотируемым параметром. Параметр `length` должен относиться к конечному значению параметра `_out`. Параметры могут быть следующими:

- ◆ опущен — размер явно не указан. Применяется, когда не используется ни `_ecount`, ни `_bcount`;
- ◆ `(size)` — общий размер буфера. Применяется с аннотациями `_ecount` или `_bcount`, но не с `_part`;
- ◆ `(size, length)` — общий размер буфера и длина его инициализированной части соответственно. Используется с аннотациями `_ecount_part` и `_bcount_part`.

## Полезные советы по аннотированию буферов

При аннотировании буферов, помните следующее.

- ◆ Каждая аннотация буфера описывает один буфер, с которым взаимодействует функция: где находится буфер, его размер, размер его инициализированной части, а также каким образом функция использует его.

Буфером может быть строка, массив фиксированной или переменной длины либо просто указатель.

- ◆ Для каждого параметра следует применять только одну аннотацию буфера.
- ◆ Некоторые комбинации элементарных аннотаций не имеют смысла в качестве аннотаций буфера. Список комбинаций аннотаций, имеющих смысл, см. в определениях аннотации буфера в заголовочном файле `Specstrings.h`.

## Примеры аннотаций буфера

В листингах 23.8 и 23.9 приводятся примеры аннотаций буфера.

В листинге 23.8 выполняется замена по месту (*in-place*) символов в счетном массиве. Входным значением для `*s` является старый размер, а выводом — новый размер. Эту функцию можно применить для замены не-ASCII-символов символами в кодировке UCS-8.

### Листинг 23.8. Пример аннотаций для замены по месту (*in-place*) символов в счетном массиве

```
void substUCS8(
    _inout_ecount_part(*s, *s) wchar_t *buffer,
    _inout size_t *s);
```

В листинге 23.9 показано применение аннотации `_xcount` для указания размера буфера, который не может быть представлен в виде простого выражения. В действительности, эту

функцию можно реализовать несколькими другими лучшими способами. Этот пример приводится только для иллюстрации применения аннотации `_xcount`. В зависимости от входного параметра `which`, функция возвращает строку, длина которой равна одному из трех известных значений. Данная аннотация заставляет PREfast не выполнять проверку размера буфера `*msgBuffer`.

**Листинг 23.9. Пример аннотации `_xcount` для указания размера буфера, который не может быть представлен в виде простого выражения**

```
GetString(
    _out_xcount("23, 42, or 26, depending on 'which'")
    LPSTR *msgBuffer,
    _in which);
```

### Полезная информация

Несколько сходных аннотаций для указания размера буфера определены в заголовочном файле `Specstrings.h`. Если вы не можете найти требующуюся вам аннотацию, прочитайте комментарии в этом файле, чтобы узнать, не решена ли уже ваша проблема.

## Аннотации строк

В языке С строка, оканчивающаяся завершающим нулем, представляет собой специальный вид буфера. Следующие аннотации описывают строки, оканчивающиеся завершающим нулем

- ◆ `_nullterminated;`
- ◆ `_nullnullterminated;`
- ◆ `_possibly_notnullterminated.`

Строковые аннотации полезны при использовании с объявлениями `typedef`. Эти аннотации позволяют PREfast проверить правильность использования типа в функции, не требуя, чтобы программист аннотировал в функции каждый параметр данного типа. Для получения дополнительной информации об объявлениях функций `typedef` см. разд. "Аннотации к объявлениям `typedef` функций" ранее в этой главе.

### Примечание

Примеры в этом разделе предназначены только для иллюстрации использования аннотаций для строк, оканчивающихся завершающим нулем. Вместо того чтобы создавать собственные функции для манипуляций строками и строками типа `UNICODE_STRING`, всегда следует применять безопасные строковые функции, объявленные в заголовочном файле `%wdk%\inc\ddk\ntstrsafe.h`.

### Аннотация `_nullterminated`

Многие из типов, объявленных в системных заголовочных файлах, уже аннотированы. Если для всех функций, принимающих в качестве параметров строки типа `char*` или `wchar_t*`, используется соответствующий тип `STR`, применение для этих типов аннотации `_nullterminated` не является обязательным. Типы `PSTR` и `PCSTR` и их варианты `L` и `W` подразумевают, что строка оканчивается завершающим нулем. Аннотацию `_nullterminated` следует явно применять только к тем типам, к которым эта аннотация еще не была применена.

Для входных строковых буферов неявное применение аннотации `_nullterminated` посредством типов `PSTR` или `PCSTR` является достаточным. Если параметр предназначен строго для ввода, используйте формы `CSTR`, т. к. модификатор `const` должен вставляться в объявлении `typedef`.

Если функция может создавать строку или добавлять к ней символы, то для нее требуется действительный размер выходного буфера, чтобы избежать его переполнения. Выходные буфера также должны иметь дополнительную аннотацию размера `_ecount` или `_bcount`, которая предоставляет действительный размер буфера, т. к. аннотация `_nullterminated` сама по себе не предоставляет эту информацию. Для параметров `_out` аннотация числа байтов или элементов указывает, что последняя строка заканчивается завершающим нулем и что PREfast должен выполнять проверки на предмет переполнения буфера. Для параметров `_inout` аннотация числа байтов или элементов подразумевает, что буфер инициализируется вплоть до `NULL`, и что обновленное значение также оканчивается завершающим нулем.

Рассмотрим, например, функцию `StringCchCopy`, которая копирует вплоть до `cchDest` элементов. Аннотация `_out_ecount` указывает, что хотя строка и заканчивается завершающим нулем (это исходит из типа `LPSTR`), она не выходит за пределы `cchDest` байтов, как показано в следующем коде:

```
StringCchCopyA(
    _out_ecount(cchDest) LPSTR pszDest,
    _in size_t cchDest,
    _in LPCSTR pszSrc);
```

### **Аннотация `_nullnullterminated`**

Аннотация `_nullnullterminated` предназначена для нечасто встречающейся "строки строк", оканчивающейся двойным завершающим нулем, например, параметр реестра типа `REG_MULTI_SZ`. В настоящее время PREfast не проверяет `_nullnullterminated`, поэтому эта аннотация должна рассматриваться только в качестве информационной. Но она может быть задействована в одной из будущих версий PREfast. А тем временем, для подавления шума в результатах анализов PREfast, связанного со строками, оканчивающимися двойным `NUL`, используйте директиву `#pragma warning` или аннотацию `_xcount`.

### **Аннотация `_possibly_notnullterminated`**

Несколько устарелых функций обычно возвращают строки, оканчивающиеся завершающим нулем, но иногда возвращаемые ими строки не оканчиваются нулем. Классическим примером таких функций являются функции `sprintf` и `strncpy`, которые опускают конечный ноль, если значащая строка занимает точно отведенный ей буфер. Эти функции считаются устарелыми (`deprecated`), и их не следует применять. Вместо них для приложений пользовательского режима нужно применять эквивалентные функции, объявленные в заголовочном файле `StrSafe.h`, а для кода режима ядра — функции, объявленные в заголовочном файле `NtStrSafe.h`. Успешное завершение этих функций гарантирует завершение буфера конечным нулем.

Но полностью выполнить такую замену в существующем коде может быть невозможно, поэтому в таких функциях следует применить аннотацию `_possibly_notnullterminated` к их выходным параметрам, как показано в следующем примере:

```
int _snprintf(
    _out_ecount(count) _possibly_notnullterminated LPSTR buffer,
```

```
_in size_t count,
_in LPCSTR *format
[, argument] ...
);
```

Когда PREfast обнаруживает аннотацию `_possibly_notnullterminated`, он пытается определить, было ли предпринято какое-либо действие, чтобы обеспечить нулевое окончание выходной строки. Если он не находит признаков такого действия, то PREfast выдает предупреждение.

## Зарезервированные параметры

Некоторые функции имеют параметры, предназначенные для будущего использования. В случае такой функции, применение аннотации `_reserved` обеспечивает, что будущие версии функции станут уверенно распознавать вызывающих клиентов для старой версии. Эта аннотация требует, чтобы предоставляемый параметр был 0 или NULL, в зависимости от того, что требуется для данного типа.

Например, в будущем следующая функция может принимать настоящее значение для второго параметра, но в настоящее время этот параметр должен быть представлен в коде как NULL. Как показано далее, применение данной аннотации позволяет PREfast проверить, чтобы текущие клиенты, вызывающие функцию, не использовали зарезервированный параметр неправильным образом:

```
void do_stuff(struct a *pa, _reserved void *pb);
```

## Возвращаемые функциями значения

Многие функции возвращают значение статуса, указывающее, было ли выполнение функции успешным. Но довольно распространенной практикой является применение кода, в котором всегда предполагается успешное выполнение функции, и не выполняется никакой проверки возвращаемого значения статуса исполнения. Часто таким кодом является код для выделения памяти, но и другие виды кода также грешат этим. Так, функция `malloc` является классическим примером функции, для которой следует применять аннотацию `_checkReturn`, как показано в следующем примере:

```
_checkReturn void *malloc(_in size_t s);
```

Эта аннотация указывает, что необходимо выполнить проверку возвращаемого функцией значения. Для функций с аннотациями `_checkReturn` PREfast может выявить ошибки двух типов:

- ◆ возвращаемое функцией значение просто игнорируется;
- ◆ возвращаемое функцией значение помещается в переменную, которая потом игнорируется.

Чтобы избежать получения предупреждения при вызове функции с аннотацией `_checkReturn`, необходимо либо использовать возвращаемое значение непосредственно в условном выражении, либо присвоить его переменной, которую затем использовать в условном выражении. Хотя аннотация `_checkReturn` обычно применяется с возвращаемыми значениями, PREfast может выявить применение этой аннотации с параметром `_out` и требовать выполнения проверки значения этого параметра.

В редких случаях, когда необходимо игнорировать возвращаемое значение, функцию следует вызывать в явном контексте `void` следующим образом:

```
(void)mustCheckReturn(param)
```

Возвращение значения вызывающему клиенту рассматривается как успешная проверка возвращаемого значения. Но к самому возвращенному параметру или значению необходимо применить аннотацию `_checkReturn`, чтобы вызывающий клиент мог бы проверить возвращенное значение.

Для драйверов режима ядра необходимо аннотировать все операции выделения памяти, и драйвер должен попытаться завершить неудачную операцию выделения памяти наиболее организованным образом.

### Примечание

Если вы когда-либо пользовались опцией `/analyze` в Visual Studio, то, может быть, заметили, что при анализе функций с аннотациями `_checkReturn` PREfast работает несколько по-другому. Как опция `/analyze`, так и PREfast выдают предупреждение в случае игнорирования возвращаемого значения функции в точке вызова функции. Но PREfast также выдает предупреждение, если возвращаемое значение функции присваивается переменной, которая затем не используется в последующем коде.

## Аннотации для драйверов

Аннотации для драйверов расширяют возможности PREfast по нахождению ошибок, специфичных для исходного кода драйверов. Эти аннотации предназначены для дополнения аннотаций общего назначения, рассмотренных ранее в этой главе. Аннотации для драйверов определены в заголовочном файле `%wdk%\inc\ddk\Driverspecs.h`; их имена начинаются с префикса `_drv`.

Хотя аннотации для драйверов изначально разрабатывались для драйверов режима ядра WDM, с некоторыми исключениями, они применяются на достаточно низком уровне, поэтому их можно использовать с любой драйверной моделью. Многие аннотации для драйверов можно также применять для кода режима ядра общего назначения или драйверов UMDF. Большинство примеров в этом разделе взято из драйверов WDM, но иллюстрируемое ими применение также должно подойти и для драйверов WDF. Некоторые аннотации имеют смысл только для кода режима ядра и, поэтому, не имеют смысла для драйверов UMDF. В этой главе на такие аннотации обращается внимание.

В аннотациях для драйверов используется немного иной синтаксис, чем для аннотаций общего назначения, таких как `_in` и `_out`. Некоторые аннотации для драйверов идентифицируют точный объект, к которому они применяются. Например, аннотация `_drv_arg` применяет список аннотаций для именованного формального параметра функции. Другие аннотации для драйверов описывают семантику функции. Например, аннотация `_drv_MaxIRQL` указывает максимальный уровень прерываний, на котором функция может вызываться.

Все аннотации для драйверов применяются явно или неявно к определенному элементу аннотируемой функции. Определенным элементом может быть один из следующих:

- ◆ сама функция (т. е. глобальное состояние функции);
- ◆ параметр функции;
- ◆ возвращаемое функцией значение;
- ◆ указатель `this` в коде C++.

Аннотации для драйверов можно применять на любом уровне разыменования, как и частичную аннотацию общего назначения `_derefer`. Аннотация может относиться к предыдущему или последующему состоянию функции или параметра.

### Внимание!

Аннотации общего назначения низкого уровня, такие как `_notnull`, можно комбинировать с аннотациями для драйверов. Но составные аннотации общего назначения, содержащие такие элементарные аннотации, как `_in`, `_out` и `_inout`, с аннотациями для драйверов, комбинировать нельзя. Например, составная аннотация общего назначения `_inout_ecount_part` должна стоять отдельно от аннотаций для драйверов — или рядом с ними, или на отдельной строке — и ее нельзя конкатенировать с аннотацией для драйверов или включать в список аннотаций.

В табл. 23.3 приведен список аннотаций для драйверов в том порядке, в каком они рассматриваются в этой главе.

**Таблица 23.3. Аннотации для драйверов**

| Аннотация                                                                                                                                                                                                                                                                                                                                                    | Применение                                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| <code>_drv_arg(org, anno_list)</code><br><code>_drv_arg(_param(n), anno_list)</code><br><code>_drv_deref(anno_list)</code><br><code>_drv_fun(anno_list)</code><br><code>_drv_in(anno_list)</code><br><code>_drv_in_deref(anno_list)</code><br><code>_drv_out(anno_list)</code><br><code>_drv_out_deref(anno_list)</code><br><code>_drv_ret(anno_list)</code> | Базовые аннотации для драйверов                      |
| <code>_drv_when(cond, anno_list)</code>                                                                                                                                                                                                                                                                                                                      | Условные аннотации                                   |
| <code>_drv_valueIs(list)</code>                                                                                                                                                                                                                                                                                                                              | Аннотации результатов функций                        |
| <code>_drv_strictTypeMatch(mode)</code><br><code>_drv_strictType(typename, mode)</code>                                                                                                                                                                                                                                                                      | Аннотации типов                                      |
| <code>_drv_notPointer</code><br><code>_drv_isObject Pointer</code>                                                                                                                                                                                                                                                                                           | Аннотации указателей                                 |
| <code>_drv_constant</code><br><code>_drv_nonConstant</code>                                                                                                                                                                                                                                                                                                  | Аннотации для постоянных и переменных параметров     |
| <code>_drv_formatString(kind)</code>                                                                                                                                                                                                                                                                                                                         | Аннотации строк форматирования                       |
| <code>_drv_preferredFunction(name, reason)</code><br><code>_drv_reportError(string)</code>                                                                                                                                                                                                                                                                   | Диагностические аннотации                            |
| <code>_drv_inTry</code><br><code>_drv_notInTry</code>                                                                                                                                                                                                                                                                                                        | Аннотации для функций в операторах <code>_try</code> |
| <code>_drv_allocatesMem(type)</code><br><code>_drv_freesMem(type)</code><br><code>_drv_aliasesMem</code>                                                                                                                                                                                                                                                     | Аннотации для памяти                                 |

Таблица 23.3 (окончание)

| Аннотация                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Применение                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|
| <code>_drv_acquiresResource(kind)</code><br><code>_drv_releasesResource(kind)</code><br><code>_drv_acquiresResourceGlobal(kind, param)</code><br><code>_drv_releasesResourceGlobal(kind, param)</code><br><code>_drv_acquiresCriticalSection</code><br><code>_drv_releasesCriticalSection</code><br><code>_drv_acquiresCancelSpinLock</code><br><code>_drv_releasesCancelSpinLock</code><br><code>_drv_mustHold(kind)</code><br><code>_drv_neverHold(kind)</code><br><code>_drv_mustHoldGlobal(kind, param)</code><br><code>_drv_neverHoldGlobal(kind, param)</code><br><code>_drv_mustHoldCriticalSection</code><br><code>_drv_neverHoldCriticalSection</code><br><code>_drv_mustHoldCancelSpinLock</code><br><code>_drv_neverHoldCancelSpinLock</code><br><code>_drv_acquiresExclusiveResource(kind)</code><br><code>_drv_releasesExclusiveResource(kind)</code><br><code>_drv_acquiresExclusiveResourceGlobal(kind, param)</code><br><code>_drv_releasesExclusiveResourceGlobal(kind, param)</code> | Аннотации для ресурсов иных, чем память |
| <code>_drv_functionClass</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Аннотации класса функций                |
| <code>_drv_floatSaved</code><br><code>_drv_floatRestored</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Аннотации для плавающей запятой         |
| <code>_drv_maxIRQL(value)</code><br><code>_drv_minIRQL(value)</code><br><code>_drv_setsIRQL(value)</code><br><code>_drv_requiresIRQL(value)</code><br><code>_drv_raisesIRQL(value)</code><br><code>_drv_savesIRQL</code><br><code>_drv_restoresIRQL</code><br><code>_drv_savesIRQLGlobal(kind, param)</code><br><code>_drv_restoresIRQLGlobal(kind, param)</code><br><code>_drv_minFunctionIRQL(value)</code><br><code>_drv_maxFunctionIRQL(value)</code><br><code>_drv_sameIRQL</code><br><code>_drv_isCancelIRQL</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Аннотации IRQL                          |
| <code>_drv_clearDoInit</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Аннотации DO_DEVICE_INITIALIZING        |
| <code>_drv_interlocked</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Аннотации для операнда с блокировкой    |

## Базовые аннотации и соглашения по их применению

Аннотации, перечисленные в табл. 23.4, можно применять в любом месте драйвера. Аргумент `anno_list` состоит из одной или нескольких аннотаций, разделенных пробелами.

**Таблица 23.4. Базовые аннотации для драйверов**

| Аннотация                                                                            | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_drv_arg(arg, anno_list)</code><br><code>_drv_arg(_param(n), anno_list)</code> | Указывает, что аннотации в параметре <code>anno_list</code> относятся к параметру <code>arg</code> , который является именованным формальным параметром функции. Аргумент <code>arg</code> может быть либо именем параметра, либо указателем <code>this</code> языка C++. С помощью оператора разыменования <code>*</code> можно указать уровень разыменования, к которому относится аннотация. Вместо <code>arg</code> можно применить аннотацию <code>_param(n)</code> , чтобы указать позицию параметра (начиная с базовой позиции 1). Например, <code>_param(3)</code> указывает, что аннотация относится к третьему параметру функции |
| <code>_drv_deref(anno_list)</code>                                                   | Указывает, что к аннотации необходимо применить дополнительный уровень разыменования. Эта аннотация эквивалентна аннотации <code>_drv_arg(*_param(n), anno_list)</code> , т. е. она относится к разыменованному значению <code>n</code>                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>_drv_fun(anno_list)</code>                                                     | Указывает, что аннотации в параметре <code>anno_list</code> относятся ко всей функции. То есть, аннотация относится к какому-либо свойству глобального состояния вызывающей функции                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>_drv_in(anno_list)</code>                                                      | Указывает, что <code>anno_list</code> относится к вводу — предусловие                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>_drv_in_deref(anno_list)</code>                                                | Эквивалентна аннотации <code>_drv_in(_drv_deref(anno_list))</code> . Эта аннотация предоставлена для удобства                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>_drv_out(anno_list)</code>                                                     | Указывает, что <code>anno_list</code> относится к выводу — постусловие                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>_drv_out_deref(anno_list)</code>                                               | Эквивалентна аннотации <code>_drv_out(_drv_deref(anno_list))</code> . Эта аннотация предоставлена для удобства                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>_drv_ret(anno_list)</code>                                                     | Указывает, что аннотации в параметре <code>anno_list</code> относятся к возвращаемому функцией значению                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

Списки аннотаций и вложенные аннотации создаются согласно соглашениям, описанным далее.

### Списки аннотаций для драйверов

Список аннотаций может содержать одну или несколько следующих аннотаций:

- ◆ аннотации позиций, такие как, `_deref`, `_drv_deref` или `_drv_arg`;
- ◆ условные аннотации, задаваемые префиксом `_drv_when`;
- ◆ аннотации общего назначения, такие как `_notnull`;
- ◆ другие аннотации, которые имеют смысл для функции или параметра.

Списки аннотаций часто используются с аннотацией `_drv_arg`, что помещает аннотации, относящиеся к параметру функции, в начале функции, а не в строку кода, содержащей параметр. Например, следующая аннотация `_drv_arg` относится к параметру `NumberOfBytes`. Список аннотаций содержит только одну аннотацию, `_in`, как показано в следующем примере:

```
_drv_arg(NumberOfBytes, in)
```

В следующем примере показана аннотация `_drv_arg`, которая относится к параметру `*pBuffer` функции:

```
_drv_arg(*pBuffer, _drv_neverHold(EngFloatState) _drv_notPointer)
```

Список аннотаций содержит две аннотации: аннотацию `_drv_neverHold(EngFloatState)`, которая указывает, что при вызове функции не должно удерживаться состояние плавающей запятой, и аннотацию `_drv_notPointer`, которая указывает, что `*pBuffer` должен указывать непосредственно в память. Более подробно эти аннотации объясняются в разд. "Примеры аннотированных системных функций" далее в этой главе. Также см. листинг 23.32.

Другими аннотациями для драйверов, принимающими списки аннотаций, являются аннотации `_drv_in` и `_drv_out`. Например, следующий список аннотаций указывает, что как аннотированный выходной параметр, так и его первый уровень разыменования не может быть `NULL`. Этот список аннотаций помещается в строку кода, содержащего параметр функции, следующим образом:

```
_drv_out(_notnull _deref(_notnull))
```

Обратите внимание на использование аннотации `_deref`, накладывающее вторую аннотацию `_notnull` к разыменованному значению параметра.

Списки аннотаций могут стать громоздкими, особенно если аннотации применяются на разных уровнях разыменования. Например, допустим, требуется аннотировать параметр `***p`, как входной параметр, который не может быть `NULL`.

Если поместить необходимые элементарные аннотации в одну строку, конечная составная аннотация будет выглядеть следующим, очень неудобочитаемым, образом:

```
_drv_in(_drv_deref(_drv_deref(_drv_deref(_notnull)))) char ***p
```

Обратите внимание на три вложенные аннотации `_drv_deref`, которые требуются для применения аннотации `_notnull` с `***p`.

Использование аннотации `_drv_arg` делает аннотацию намного более удобочитаемой, как можно видеть в следующем примере:

```
_drv_arg(***p, _drv_in(_notnull)) char ***p
```

Данная аннотация `_drv_arg` ассоциирует аннотацию с `***p`, вместо тройного использования аннотации `_drv_deref`, чтобы разыменовать `***p`.

## Вложенные аннотации

Базовые аннотации для драйверов можно комбинировать путем вложения. Так как одну и ту же аннотацию можно выразить несколькими способами, решение о том, какой из способов применить, обычно делается на основе удобочитаемости конечной аннотации.

В следующих пяти примерах делается абсолютно одно и то же: аннотации в списке `anno_list` применяются к параметру `p1` типа `xyz` на одном уровне разыменования. В этих примерах демонстрируется, как создавать составные аннотации и как отделять сложные аннотации от параметра, к которому они относятся.

В следующих двух примерах аннотации в списке `anno_list` применяются к параметру `p1` типа `xyz` на одном уровне разыменования. Обе эти аннотации нужно поместить в строку кода, в которой объявляется параметр:

```
_drv_in_deref(anno_list) xyz p1;
_drv_in(_drv_deref(anno_list)) xyz p1;
```

Следующие три примера эквивалентны двум предыдущим, но их можно поместить в любом месте функции, в котором аннотация для этой функции будет действительной. Размещать их возле объявления `p1` не является обязательным:

```
_drv_arg(p1, _drv_in(_drv_deref(anno_list)))
_drv_arg(*p1, _drv_in(anno_list))
_drv_arg(p1, _drv_in_deref(anno_list))
```

Для получения дополнительной информации по вложенным аннотациям см. разд. "Примеры аннотированных системных функций" далее в этой главе.

## Условные аннотации

Некоторые функции имеют сложные интерфейсы, определенный аспект которых действует только в конкретных ситуациях. Безусловное применение аннотаций с такими функциями может создать ситуацию, в которой PREfast будет выдавать какие-либо ложноположительные результаты для совершенно правильного кода независимо от того, каким образом составлены аннотации.

Эту проблему можно решить условным применением аннотаций с помощью аннотации `_drv_when(cond, anno_list)`. Данная аннотация указывает, что аннотации в списке `anno_list` следует проверять только в случае выполнения условия, указанного в выражении `cond`. Значения компонентов аннотации `_drv_when` следующие.

- ◆ Выражение `cond` указывает условие, которое вычисляется в соответствии с синтаксисом языка C.

Если нельзя вычислить постоянное значение условия, то PREfast эмулирует исполнение функции как с истинной, так и с ложной возможностью.

- ◆ Список `anno_list` содержит соответствующие аннотации, как было описано в разд. "Списки аннотаций для драйверов" ранее в этой главе.

Аннотацию `_drv_when` можно свободно применять совместно с аннотациями, описанными в разд. "Базовые аннотации и соглашения по их применению" ранее в этой главе.

Например, условие в основном применяется для указания, что если вызываемая функция выполняется успешно, то она получает какой-либо ресурс — память или ресурс другого типа. Если же вызываемая функция исполняется неудачно, то она не получает этого ресурса, и, соответственно, утечки данного ресурса быть не может. Эта информация имеет особенную важность для функций, которые возвращают NTSTATUS, чтобы указать успешное выполнение, т. к. существует возможность, что получаемый ресурс не изменится каким-либо образом, который можно было бы обнаружить. Только значение статуса функции указывает на ее успешное или неточное исполнение. Для получения дополнительной информации по аннотациям для ресурсов см. разд. "Аннотации для памяти" далее в главе.

Одним из более распространенных примеров функции, извлекающей пользу из условной аннотации, является функция `ExAcquireResourceExclusiveLite`, как показано в следующем примере:

```
BOOLEAN ExAcquireResourceExclusiveLite(_in PERESOURCE Resource,
   _in BOOLEAN Wait);
```

Если значение параметра `Wait` истинно, то возвращаемое функцией значение BOOLEAN проверять не нужно. Но если значение `Wait` ложно, то возвращаемое значение нужно проверять во

всех случаях. Одним из подходов было бы написать код, который проверяет, когда `Wait` истинно, но это было и раздражающим и сбивающим с толку. Но невыполнение проверки параметра `Wait` на ложность может вызвать серьезную ошибку.

В случае, когда `Wait` ложно, для выполнения качественного анализа PRE/fast требуется аннотация `_checkReturn`. А чтобы не проверять возвращаемое значение, когда значение `Wait` истинно, применяется аннотация `_drv_when`, чтобы сделать выполнение аннотации `_checkReturn` условным. Пример использования этой аннотации показан в листинге 23.10.

#### Листинг 23.10. Пример использования условной аннотации

```
_drv_when(!Wait, _checkReturn)
BOOLEAN ExAcquireResourceExclusiveLite(
    _in PERESOURCE Resource,
    _in BOOLEAN Wait);
```

### Примеры вложенных условных аннотаций

В следующем примере показано вложение нескольких аннотаций для драйверов в одном условии. Аннотации указывают, что когда значение `Wait` истинно, то `p` не может быть `NULL`.

```
_drv_when(Wait, _drv_arg(p, _drv_in(_drv_deref(_notnull)))
```

Начиная с конца, значения аннотаций в предыдущем примере следующие.

- ◆ Составная аннотация `_drv_in(_drv_deref(_notnull))` является списком аннотаций, которые необходимо применить к `p`. Эти аннотации указывают, что `p` необходимо проверить на входе и на выходе, и что разыменованное значение `p` не может быть `NULL`.
- ◆ Объемлющая аннотация `_drv_arg` идентифицирует `p` как аргумент, ассоциированный со списком аннотаций.
- ◆ Объемлющая аннотация `_drv_when` указывает, что когда значение `Wait` истинно, то к `p` нужно применить аннотации в `_drv_arg`.

Аннотации в `_drv_when` могут быть вложенными. Внутренняя аннотация применяется, когда и внутреннее, и внешнее условие истинно. В следующем примере показано вложение двух условных аннотаций. Эти аннотации указывают, что когда значение `Wait` истинно, то `p` не может быть `NULL`, и если и `Wait` и `p2` истинны, то `p3` не может быть `NULL`:

```
_drv_when(Wait, _drv_arg(p, _drv_in(_notnull))
    _drv_when(p2, _drv_arg(p3, _drv_in(_notnull)))
```

Начиная с конца второй строки этого примера — внутреннее условие:

- ◆ составная аннотация `_drv_arg(p3, _drv_in(_notnull))` указывает аннотации, которые применяются к `p3`;
- ◆ аннотация `_drv_in(_notnull)` представляет собой список аннотаций, который описывает входной аргумент, который не может быть `NULL`;
- ◆ объемлющая аннотация `_drv_arg(p3...)` идентифицирует `p3` как аргумент, ассоциированный со списком аннотаций;
- ◆ объемлющая аннотация `_drv_when(p2, ...)` указывает, что аннотации следует применять к `p3`, только если `p2` истинно.

Начиная с конца первой строки этого примера — внешнее условие:

- ◆ составная аннотация `_drv_arg(p, _drv_in(_notnull))` указывает аннотации, которые применяются к `p3`;
- ◆ аннотация `_drv_in(_notnull)` является списком аннотаций. Аннотации в этом списке описывают входной аргумент, который не может быть `NULL`;
- ◆ объемлющая аннотация `_drv_arg(p ...)` ассоциирует `p3` со списком аннотаций;
- ◆ объемлющая аннотация `_drv_when( Wait...)` указывает, что аннотации следует применять к `p` только тогда, когда и `Wait`, и `p2` оба истинны (т. е. `Wait && p2`).

## Грамматика условных выражений

Грамматика условных выражений в аннотациях является подмножеством грамматики, поддерживаемой языком программирования C.

### Поддерживаемая грамматика условных выражений

Грамматика условных выражений поддерживает следующее:

- ◆ операторы `+`, `-`, `*`, `/`, унарный `-`, `<<`, `>>`, `<`, `<=`, `==`, `!=`, `>`, `>=`, `!`, `&&`, `||`, `~`, `&`, `|`, `?::`, ( и ). PREfast поддерживает эти операторы с обычным предшествованием, а также функции семейства `$`, которые рассматриваются в разд. "Специальные функции в условных выражениях" далее в этой главе;
- ◆ все формы целочисленных констант;
- ◆ имена параметров и унарный оператор `*` с именами параметров.

### Вычисление выражений

При вычислении выражения PREfast действует следующим образом.

- ◆ Сворачивает константы.

Иными словами, когда PREfast может определить значение параметра, то он использует это значение. Так же, как это делается в языке C, PREfast преобразовывает булевые выражения в нулевое или ненулевое значение. В выражениях можно употреблять имена констант, определенных в выражениях `#define`.

- ◆ Выполняет все вычисления в формате широкого целого числа со знаком.  
PREfast не полагается на неявное усечение к более узкому размеру слова.
- ◆ Вычисляет выражения в каждой аннотации независимо.

В зависимости от контекста, для данной функции все, некоторые или ни одно из условных выражений могут быть истинны. Если два условия должны быть взаимоисключающими, их нужно кодировать таким образом, чтобы они в действительности были такими.

### Неподдерживаемые выражения

PREfast не поддерживает следующие выражения:

- ◆ перечислимые константы;
- ◆ константы типа `const` (в C++);
- ◆ оператор `sizeof` в условиях.

В зависимости от обстоятельств, часто можно найти обходное решение этих ограничений путем комбинирования имеющихся аннотаций.

### Специальные функции в условных выражениях

Из условных выражений можно вызывать определенные функции. PREfast исполняет эти функции, когда он анализирует код с содержащими их аннотациями.

```
strstr$(p1, p2)
macroDefined$(p1)
isFunctionClass$(name)
```

◆ Функция `strstr$(p1, p2)`.

Эта функция вычисляет то же самое значение, что и функция `strstr` языка C, за исключением того, что результатом является смещение в строке `p1`, по которому находится первое вхождение подстроки `p2`. Если `p1` не содержит `p2`, то функция `strstr$` возвращает `-1`. Аргументы `p1` и `p2` могут быть или выражения параметра или литералы.

Функция обычно применяется, чтобы определить (в пределах возможностей PREfast), содержит ли `p1` константу `p2`. Например, в следующем вызове функции определяется, содержит ли путь символы обратной косой черты:

```
strstr$(path, "\\")>=0
```

А в следующем вызове функции определяется, начинается ли путь с символов "C:":

```
strstr$(path, "C:")==0
```

◆ Функция `macroDefined$(p1)`.

Эта функция определяет, является ли `p1` символом, определенным с помощью `#define`. Параметр `p1` должен быть строкой в кавычках. Если `p1` не является определенным символом, то функция возвращает 0, а если является — то 1. Определенные символы можно использовать непосредственно в аннотациях условий `_drv_when`, но функция `macroDefined$` является единственным способом для `_drv_when` определить, был ли символ определен.

Функция `macroDefined$` функционально эквивалентна макросу `_drv_defined`, который имеет такой же синтаксис, как и функция `macroDefined$`, но обрабатывает строки в кавычках более надежным образом. Подобно функции `macroDefined$` макрос `_drv_defined(XYZ)` возвращает булево значение, которое указывает, является ли `XYZ` определенным символом в программе.

Аннотацию часто можно выразить, используя или `_drv_defined`, или `macroDefined$`, просто предполагая, что макрос разворачивается компилятором. Если этот подход не работает, используйте `_drv_defined`.

◆ Функция `isFunctionClass$(name)`.

Эта функция определяет, относится ли аннотируемая функция к классу, идентифицируемому параметром `name`.

## Аннотации результатов функций

Многие функции имеют ограниченный набор возможных результатов выходного параметра или возвращаемого значения. Предоставление PREfast информации об этом часто выливает-

ся в более аккуратные результаты анализов, т. к. PREfast может избежать рассмотрения невозможных ветвлений. Например, функция, возвращающая булево значение, возвращает целое число, которое может иметь любое значение. Булев тип является просто соглашением, которое указывает, что значение должно быть или истиной (TRUE), или ложью (FALSE).

С помощью аннотации `_drv_valueIs(list)` можно указать набор возможных возвращаемых значений функции. Значение выходного параметра или возвращаемое функцией значение должно быть одним из значений в параметре `list`, который содержит последовательность частичных выражений в формате `<оператор отношения><константа>`, разделенных точками с запятыми.

Рассмотрим для примера следующий код:

```
BOOLEAN b = boolfunc(...);
if (b == TRUE) {
    // Выполняется одна операция.
}
...
if (b == FALSE) {
    // Выполняется другая операция.
}
```

PREfast рассматривает `b` как целое число, но он не может определить, что `b` может иметь только два возможных значения. Поэтому во время анализа этой функции PREfast может эмулировать ситуации, в которых оба действия пропускаются, например, если PREfast предположит, что `b` равно 3. Такие ситуации могут вызывать как ложноположительные, так и ложноотрицательные результаты анализов. В этом примере к параметру `b` можно применить аннотацию `_drv_valueIs(==0; ==1)`, которая ограничивает возможный диапазон значений `b` к значениям TRUE и FALSE.

Для функций, возвращающих NTSTATUS, обычно применяется аннотация `_drv_valueIs(<0;==0)`, которая указывает диапазон значений для успеха и неудачи.

А для некоторых функций может быть лучше применять аннотацию `_drv_valueIs(<0;>=0)`. Для NTSTATUS эти функции могут возвращать только положительные значения, что означает, что набор возможных результатов состоит только из целых чисел — меньше нуля и больше или равных нулю. Эта аннотация, которая, в сущности, совсем не аннотация, рассматривается здесь только для полноты представления.

Условия можно комбинировать с аннотацией `_drv_valueIs`, чтобы ограничить значения результата значениями, возможными для данных входных параметров. Например, аннотации, которые применяются к возвращаемому значению функции `ExAcquireResourceExclusiveLite`, указывают, что если параметр `Wait` равен 0, функция может возвращать или 0 или 1. Но если значение `Wait` не равно 0, то функция может возвращать только 1, как показано в следующем примере:

```
_drv_when(!Wait, _drv_valueIs(==0; ==1))
_drv_when(Wait, _drv_valueIs(==1))
```

Альтернативная аннотация может выглядеть следующим образом:

```
_drv_when(!Wait, _drv_valueIs(==0;==1) _checkReturn)
```

Данная аннотация указывает, что если `Wait` ложно, то функция возвращает 0 или 1, и результат необходимо проверить. Но если `Wait` не равно нулю, возвращаемое функцией значение не важно, т. к. его проверять не требуется.

## Аннотации для сопоставления типов

В языках С и C++ разрешается определенная степень смешения типов целых чисел. В частности, разрешается передавать перечислимый тип, когда ожидается обобщенный целочисленный тип. Но для некоторых функций легко передать неправильный перечислимый тип.

Чтобы обеспечить выполнение PREfast проверки вызова функции с параметрами правильного типа, можно применить аннотации `_drv_strictTypeMatch(mode)` и `_drv_strictType(typename, mode)` следующим образом:

- ◆ для аннотации `_drv_stringTypeMatch` фактический параметр должен быть такого же типа, что и формальный параметр, в пределах, установленных параметром `mode`;
- ◆ для аннотации `_drv_strictType` параметр должен быть типа, указанного параметром `typename`, в пределах, установленных параметром `mode`.

Параметр `mode` может быть одним из следующих:

- ◆ `_drv_typeConst` — параметр принимает один простой операнд, который должен соответствовать точным образом. Эта аннотация обычно применяется с одиночной константой;
- ◆ `_drv_typeCond` — аннотированное выражение может использовать оператор `?:`, чтобы выбирать между отдельными operandами. Также разрешается применение скобок. Эта аннотация обычно применяется, чтобы динамически переключаться между несколькими аргументами-константами;
- ◆ `_drv_typeBitset` — параметр может принимать выражения, содержащие operandы только данного типа. Эта аннотация обычно применяется для установки битов, но ее применение не ограничено битовыми операциями;
- ◆ `_drv_typeExpr` — параметр может принимать те же самые operandы, что и аннотация `_drv_typeBitset`, а также литерные константы. Например, параметр `POOL_TYPE` функции `ExAllocatePool` использует эту аннотацию.

Аннотация `_drv_strictType` полезна для функций, которые принимают целочисленные параметры, чье значение должно быть ограничено членами определенного перечислимого типа. Если функция может быть передана переменная или константа, может быть полезным задать как имя `typedef` для переменных, так и перечислимый тип для констант. Это можно сделать, задавая имя `typedef` и имя перечислимого типа, разделенные косой чертой, следующим образом:

```
_drv_strictType(KPROCESSOR_MODE/enum _MODE, _drv_typeCond)
```

Функция `KeWaitForMultipleObjects` предоставляет несколько примеров аннотаций типа. Эта функция имеет три параметра, все разных перечислимых типов. Но параметры легко перепутать, а компилятор С не проверяет на правильность тип параметра. С применением соответствующих аннотаций PREfast может проверить, чтобы параметры были требуемого типа. Для получения дополнительной информации см. примечания после листинга 23.11.

**Листинг 23.11. Пример аннотаций типа, применяемый с функцией KeWaitForMultipleObjects**

```
NTSTATUS KeWaitForMultipleObjects(
    _in ULONG Count,
    _in PVOID Object[],
    _in _drv_strictTypeMatch(_drv_typeConst) // 1
    WAIT_TYPE WaitType,
```

```
_in _drv_strictTypeMatch(_drv_typeConst) // 2
    KWAIT_REASON WaitReason,
_in _drv_strictType(KPROCESSOR_MODE/enum _MODE,
    _drv_typeCond) // 3
    KPROCESSOR_MODE WaitMode,
_in BOOLEAN Alertable,
_in_opt PLARCE_INTECER Timeout,
_in_opt PKWAIT_BLOCK WaitBlockArray;
```

Значения пронумерованных комментариями аннотаций в листинге 23.11 следующие:

1. Аннотация `_drv_strictTypeMatch(_drv_typeConst)` указывает, что переменная `WaitType` должна быть членом перечислимого типа `WAIT_TYPE`.
2. Аннотация `_drv_strictTypeMatch(_drv_typeConst)` указывает, что переменная `WaitReason` должна быть членом перечислимого типа `KWAIT_REASON`.
3. Аннотация `_drv_strictType(KPROCESSOR_MODE/enum_MODE, _drv_typeCond)` указывает, что переменная `WaitMode` должна быть или типа `KPROCESSOR_MODE`, или членом перечислимого типа `_MODE` и может быть передана, как выражение, которое использует оператор `?::`.

Относительно параметра `WaitMode` необходимо принять во внимание следующие обстоятельства.

- ◆ Константы типа `enum_MODE` семантически обоснованы, и также является семантически обоснованным желание иметь возможность выбирать их с помощью оператора `?::`. Но разрешать арифметические операции на этих константах не является обоснованным. Поэтому аннотация `_drv_typeCond` позволяет использование оператора `?::`, но не позволяет применения никаких других операторов.
- ◆ Тип `KPROCESSOR_MODE` определен как `char`, а значения перечисления определяются языком С такими же, как и тип `int`. Таким образом, константа типа `KPROCESSOR_MODE` не может иметь символьического имени. Вместо этого, самим близким совпадением является перечисление `enum_MODE`.

## Аннотации указателей

Определенные драйверные функции принимают параметры типа `PVOID`, поэтому они могут принимать параметры нескольких типов. Распространенной ошибкой является передача параметра неправильного типа, например, передача `&pStruct` вместо `pStruct`, как намеревалось. Для любого другого типа, кроме `PVOID`, компилятор бы выявил эту ошибку. С типом `PVOID` компилятор не может выявить ошибку, т. к. не имеется информации о типе, которую проверять.

Аннотации `_drv_notPointer` и `_drv_isObjectPointer` позволяют PREfast выявить ошибки, когда параметр не может быть указателем.

- ◆ Аннотация `_drv_notPointer` указывает, что значение параметра должно быть скалярное или типа `struct`.

Например, аннотация `_drv_notPointer` указывает, что `(PVOID) 1` является приемлемым значением, а `(PVOID) &var` — нет.

Аннотация `_drv_notPointer` обычно применяется в виде `_drv_deref(_drv_notPointer)`, указывая, что параметр не должен быть указателем на указатель, а должен быть указателем на объект, не являющийся указателем, обычно структуру. Эта аннотация позволяет

PREfast выявить такую распространенную ошибку, как передача `&pStruct` вместо `pStruct`, как намеревалось, т. к. вы забыли, что был передан указатель на структуру, а не структура.

- ◆ Аннотация `_drv_isObjectPointer` указывает, что параметр должен быть указателем на объект, не являющийся указателем.

Эта аннотация является более простым эквивалентом аннотации `_drv_deref(_drv_notPointer)`.

Например, в листинге 23.12 аннотация параметра `Object` функции `KeWaitForSingleObject` заставляет PREfast выдать предупреждение, если функция вызывается с параметром `Object`, являющимся указателем на указатель.

**Листинг 23.12. Пример аннотации для функции, которую нужно вызывать с указателем на объект, не являющийся указателем**

```
NTSTATUS
KeWaitForSingleObject(
    _in _drv_isObjectPointer PVOID Object,
    _in _drv_strictTypeMatch(_drv_typeConst)
        KWAIT_REASON WaitReason,
    _in _drv_strictType(KPROCESSOR_MODE/enum _MODE, _drv_typeCond)
        KPROCESSOR_MODE WaitMode,
    _in BOOLEAN Alertable,
    _in_opt PLARGE_INTEGER Timeout
);
```

## Аннотации для постоянных и переменных параметров

Аннотации `_drv_constant` и `_drv_nonConstant` можно применять для функций, которые или требуют, или запрещают литерные константы в качестве параметров.

Например, драйверу устройства нельзя предполагать, что любой адрес порта является константой. Поэтому к различным функциям семейства `READ_PORT_XXX` необходимо применить аннотацию `_drv_nonConstant` для адреса считываемого порта. Редкие предупреждения, выдаваемые PREfast при исключениях к этому требованию, можно либо игнорировать, либо подавлять.

Рассмотрим другой пример — параметр `Wait` функции `KeSetEvent`. Хотя теоретически этот параметр может быть переменной, действия, которые должны быть выполнены до и после вызова этой функции, делают трудновыполнимой задачу ее вызова с переменной. Поэтому к параметру `Wait` применяется аннотация `_drv_constant`. А случайные предупреждения в исключительных случаях также можно либо игнорировать, либо подавлять.

## Аннотации форматирующих строк

Аннотация `_drv_formatString(kind)` указывает, что параметр, к которому она применяется, является форматирующей строкой. Параметр `kind` может быть `printf` или `scanf`, что указывает тип разрешенной форматирующей строки, а не конкретную вызываемую функцию. Иными словами, форматирующая строка следует правилам форматирования функции `printf` или `scanf`. Аннотацию `_drv_formatString` можно применять с любой функцией, подобной функциям `printf` или `scanf`.

Эта аннотация заставляет PREfast проверять, чтобы список аргументов совпадал с форматирующей строкой и чтобы не применялись, возможно, небезопасные комбинации.

В следующем примере аннотация указывает, что `format` является форматирующей строкой для функции `printf`:

```
int _snprintf(
    _out_ecount(count) _possibly_notnullterminated LPSTR buffer,
    _in size_t count,
    _in _drv_in(_drv_formatString printf) LPCSTR *format
    [, argument] ...  
);
```

## Диагностические аннотации

Иногда случаются определенные комбинации параметров, которые либо небезопасные, либо их можно выразить лучше каким-нибудь другим образом. PREfast может выявить многие из таких ситуаций с помощью аннотаций `_drv_preferredFunction(name, reason)` и `_drv_reportError(string)`. Эти аннотации обычно применяются совместно с условной аннотацией `_drv_when`. Эти аннотации следует использовать для рекомендаций и для аннотирования конкретных применений, которых нужно избегать.

Если функция не должна применяться ни при каких обстоятельствах, ее нужно пометить аннотацией `#pragma_deprecated` или `_declspec(deprecated)`, чтобы компилятор мог генерировать ошибку при компиляции.

### Аннотации для предпочтительных функций

С помощью аннотаций `_drv_preferredFunction(name, reason)` можно генерировать сообщения об ошибках. Параметр `name` может быть чем угодно, но обычно это имя предпочтительной функции, а параметр `reason` дает дополнительное объяснение, почему функция является предпочтительной.

Рассмотрим, например, две гипотетические функции — `GetResource` и `TryToGetResource`. Функция `GetResource` принимает параметр `Wait`. Когда значение `Wait` равно `TRUE` (т. е. ненулевое), вызов функции должен ожидать до тех пор, пока не будет выделен требуемый ресурс. Когда значение `Wait` равно `FALSE`, то ресурс тоже можно получить с помощью функции `GetResource`, но в этом случае использование функции `TryToGetResource` будет более эффективным. В следующем коде приводится пример аннотаций, которые бы заставили PREfast обозначить эту ситуацию:

```
_drv_when(!Wait, _drv_preferredFunction(TryToGetResource,
   "Если GetResource вызывается с Wait==FALSE, "
   "то вместо нее лучше вызвать TryToGetResource."))
```

### Аннотации для сообщений об ошибках

Аннотация `_drv_reportError(string)` заставляет PREfast генерировать предупреждение о том, что он обнаружил ошибку, описанную аннотацией. Аннотация `_drv_reportError` похожа на аннотацию `_drv_preferredFunction`, за исключением того, что она генерирует предупреждение, требующее исправления проблемы, описанной в параметре `string`.

Например, аннотацией `_drv_reportError` можно пометить неприемлемое действие, такое как попытка выполнить операцию выделения памяти с обязательным успехом одной из функций семейства `ExAllocatePool`:

```
_drv_when(PoolType&0x1f==2 || PoolType&0x1f==6,
    _drv_reportError("Запрещается выделять память из пулов "
        "с условием обязательного успеха. Неудачное завершение "
        "выделения вызовет фатальный сбой системы."))
```

## Аннотации для функций в операторах `_try`

Определенные функции всегда требуется вызывать изнутри оператора `_try` для структурной обработки исключения, в то время как другие функции никогда не должны вызываться из этого оператора.

Аннотации `_drv_inTry` и `_drv_notInTry` заставляют PREfast выполнять проверки на правильность действий внутри функций:

- ◆ аннотация `_drv_inTry` указывает, что функцию необходимо вызывать из оператора `_try`;
- ◆ аннотация `_drv_notInTry` указывает, что функцию нельзя вызывать из оператора `_try`.

Например, аннотацию `_drv_inTry` можно использовать с функциями `ProbeForRead` и `ProbeForWrite` для того, чтобы можно было уловить неудачные попытки доступа к памяти с помощью оператора `_try`.

## Аннотации для памяти

Драйверы часто используют функции специального назначения для выделения и освобождения памяти. Драйверы также часто передают память из функции таким образом, что порождает совмещение имен для памяти, что будет рассмотрено позже. Аннотации могут помочь PREfast выявлять более эффективно утечки памяти и иные проблемы с выделениями памяти и других ресурсов, таких как спин-блокировки.

Аннотации для памяти, описанные в следующих разделах, можно использовать, чтобы помочь PREfast проверять функции, выделяющие память, более аккуратно.

### Аннотации для выделения и освобождения памяти

Аннотация `_drv_allocatesMem(type)` указывает, что выходное значение выделяется или посредством параметра, или через результат функции. Параметр `type` указывает тип используемого распределителя памяти. Этот параметр является рекомендательным — PREfast не проверяет его. Но он может быть задействован в одной из будущих версий PREfast. Далее приведены рекомендуемые значения для параметра `type`:

- ◆ для `malloc` и `free` `type` должен быть `mem`;
- ◆ для оператора `new` `type` должен быть `object`.

Если выделяющая память функция указывает неудачу возвращением `NULL`, к этой функции также следует применить аннотацию `_checkReturn`.

Аннотация `_drv_freesMem(type)` указывает, что память, передаваемая в качестве входного параметра, была освобождена. В последующем состоянии PREfast предполагает, что аннотированный параметр находится в инициализированном состоянии, и до того времени, пока параметр не изменится, PREfast рассматривает дальнейшие обращения посредством фактического параметра как доступ к неинициализированной переменной. Параметр `type` должен совпадать с типом, используемым в аннотации `_drv_allocatesMem`.

## Аннотации для совмещенных имен памяти

Аннотацию `_drv_aliasesMem` можно применять к входным параметрам (включая параметры `_in` и `_out`), чтобы указать, что вызываемая функция сохраняет значение параметра в той же самой области памяти, где оно будет найдено позже и, предположительно, освобождено.

В общем, PREfast не может подтвердить, была ли память, для которой применяется совмещение имен, в действительности освобождена. После вызова функции с этой аннотацией к этой области памяти могут продолжать выполняться обращения. PREfast пытается определить несколькими разными способами, когда для памяти применяется совмещение имен, но без этой аннотации он не может определить, применяется ли совмещение имен для памяти для вызываемых функций.

Аннотация `_drv_aliasesMem` применяется для подавления ложных сообщений от PREfast о возможных утечках памяти. Для этой аннотации не применяется параметр типа, т. к. она работает одинаковым образом со всеми видами памяти.

Аннотации `_drv_freesMem` и `_drv_aliasesMem` являются взаимоисключающими. Аннотация `_drv_freesMem` указывает, что память освобождается (т. е. к памяти больше нет доступа) и PREfast утверждает это, делая недействительной переменную, содержащую указатель на освобожденную память. Аннотация `_drv_aliasesMem` просто указывает, что больше нет риска утечек памяти, но память остается действительной, и к ней могут выполняться обращения в будущем.

## Примеры аннотаций для памяти

Пример в листинге 23.13 показывает некоторые аннотации, применяемые с функциями `ExAllocatePool` и `ExFreePool`, которые являются классическим примером функций, выделяющих и освобождающих память. Для этих функций выполняются дополнительные проверки параметров, которые не показаны в данном листинге.

### Листинг 23.13. Пример аннотаций для функций, выделяющих и освобождающих память

```

checkReturn
_drv_allocatesMem(Pool)
_drv_when(PoolType&0x1f==2 || PoolType&0x1f==6,
    _drv_reportError("Запрещается выделять память из пулов "
        "с условием обязательного успеха. Неудачное завершение "
        "выделения вызовет фатальный сбой системы."))
_bcount(NumberOfBytes)
VOID
ExAllocatePoolWithTag(
    _in _drv_strictTypeMatch(_drv_typeExpr) POOL_TYPE PoolType,
    _in SIZE_T NumberOfBytes,
    _in ULONG Tag
);
NTKERNELAPI
VOID
ExFreePoolWithTag(
    _in _drv_in(_drv_freesMem(Pool)) PVOID P,
    _in ULONG Tag
);

```

В примере в листинге 23.14 функция принимает и удерживает переменную `Entry`. Иными словами, функция применяет совмещение имен для памяти, занимаемой переменной `Entry`.

Аннотация `_drv_aliasesMem` в этом примере подавляет предупреждение от PREfast о возможной утечке памяти, но оставляет Entry доступной.

#### Листинг 23.14. Пример аннотаций для функций, применяющих совмещение имен для памяти

```
VOID
InsertHeadList(
    _in PLIST_ENTRY ListHead,
    _in _drv_in(_drv_aliasesMem) PLIST_ENTRY Entry
);
```

## Аннотации для ресурсов иных, чем память

Утечка ресурсов может происходить не только с ресурсами памяти, но и с некоторыми другими ресурсами, иными, чем память, такими как, например, критические области и спин-блокировки. PREfast может обнаруживать утечки таких ресурсов точно так же, как он может выявлять утечки ресурсов памяти. Но PREfast не может применять семантику для памяти к объектам, не являющимся памятью, т. к. семантические различия между двумя типами объектов вызвали бы различные неправильные результаты анализов. Например, для ресурсов, не являющихся памятью, не существует понятия совмещения имен. В общем если для ресурса необходимо применить совмещение имен, то его лучше смоделировать как память.

В табл. 23.5 приводится список и краткое описание аннотаций для ресурсов иных, чем память. Более подробно эти аннотации описываются в разделах, следующих после таблицы.

**Таблица 23.5. Аннотации для ресурсов иных, чем память**

| Аннотация                                                                                                                                                                                                                                                                                                                                                  | Применение                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| <code>_drv_acquiresResource(kind)</code><br><code>_drv_releasesResource(kind)</code>                                                                                                                                                                                                                                                                       | Аннотации для получения и освобождения ресурсов            |
| <code>_drv_acquiresResourceGlobal(kind, param)</code><br><code>_drv_releasesResourceGlobal(kind, param)</code>                                                                                                                                                                                                                                             | Аннотации для глобальных ресурсов                          |
| <code>_drv_acquiresCriticalRegion</code><br><code>_drv_releasesCriticalRegion</code><br><code>_drv_acquiresCancelSpinLock</code><br><code>_drv_releasesCancelSpinLock</code>                                                                                                                                                                               | Аннотации для критической области и спин-блокировки отмены |
| <code>_drv_mustHold(kind)</code><br><code>_drv_neverHold(kind)</code><br><code>_drv_mustHoldGlobal(kind, param)</code><br><code>_drv_neverHoldGlobal(kind, param)</code><br><code>_drv_mustHoldCriticalRegion</code><br><code>_drv_neverHoldCriticalRegion</code><br><code>_drv_mustHoldCancelSpinLock</code><br><code>_drv_neverHoldCancelSpinLock</code> | Аннотации для удерживания и неудерживания ресурсов         |
| <code>_drv_acquiresExclusiveResource(kind)</code><br><code>_drv_releasesExclusiveResource(kind)</code><br><code>_drv_acquiresExclusiveResourceGlobal(kind, param)</code><br><code>_drv_releasesExclusiveResourceGlobal(kind, param)</code>                                                                                                                 | Составные аннотации для ресурсов                           |

## Аннотации для получения и освобождения ресурсов, иных, чем память

Для обозначения получения и освобождения ресурсов, иных, чем память, в параметре функции применяются аннотации `_drv_acquiresResource(kind)` и `_drv_releasesResource(kind)`.

Для этих аннотаций параметр `kind` указывает тип ресурса. Значения параметра `kind` для получения и освобождения ресурса должны совпадать. Параметром `kind` может быть любое имя. Имена, находящиеся в употреблении в настоящее время, включают `SpinLock`, `QueuedSpinLock`, `InterruptSpinLock`, `CancelSpinLock`, `Resource`, `ResourceLite`, `FloatState`, `EngFloatState`, `FastMutex`, `UnsafeFastMutex` и `CriticalRegion`. Несвязанные применения того же самого значения `kind` не создает конфликтов.

Например, в листинге 23.15 функция получает и освобождает ресурс с названием `SpinLock`. Это значение хранится в переменной с названием `SpinLock`. Эти идентификаторы находятся в разных пространствах имен и поэтому не конфликтуют.

**Листинг 23.15. Пример аннотации для функций, получающих и освобождающих ресурс**

```
VOID
KeAcquireSpinLock(
    _inout _drv_deref(_drv_acquiresResource(SpinLock))
    PKSPIN_LOCK SpinLock,
    _out PKIRQL OldIrql
);
VOID
KeReleaseSpinLock(
    _inout _drv_deref(_drv_releasesResource(SpinLock))
    PKSPIN_LOCK SpinLock,
    _in KIRQL NewIrql
);
```

## Аннотации для глобальных ресурсов, иных, чем память

При выполнении операций с ресурсами, иными, чем память, необходимо принимать во внимание то, что ресурс часто содержится в какой-либо переменной, но состоит из глобальной информации о состоянии, доступ к которой выполняется с помощью контекста или какого-либо идентификатора.

Для обозначения получения и освобождения ресурсов этого типа можно воспользоваться аннотациями `_drv_acquiresResourceGlobal(kind, param)` и `_drv_releasesResourceGlobal(kind, param)`.

Эти аннотации применяются ко всей функции, а не к отдельным параметрам функции. Параметры аннотации `kind` и `param` указывают тип и экземпляр ресурса соответственно.

## Аннотации для присваивания имен ресурсам

С помощью аннотации `_drv_acquiresResourceGlobal` можно создавать имена для аннотирования ресурсов, иных, чем память, которые не хранятся в переменной. Данная аннотация принимает следующие два параметра:

- ◆ параметр `kind` является строковой константой (т. е. имя класса ресурса) для типа объекта (т. е. ресурса), который нужно отслеживать;

- ◆ параметр *param* является конкретным экземпляром для отслеживания. В вызове функции предоставляется имя конкретного экземпляра (т. е. *param*).

### Аннотации для присваивания имен экземплярам ресурсов

Часто, если ресурс не хранится в переменной, для идентифицирования экземпляра требуемого ресурса применяется другая переменная. Многие аннотации для ресурсов принимают параметр, идентифицирующий выделяемый экземпляр ресурса. Этот параметр похож на параметр *size* для модификатора аннотации *\_ecount(size)* в том смысле, что аннотация не применяется к самому параметру. Вместо этого значение параметра модифицирует аннотацию какого-либо другого параметра или же всей функции.

Рассмотрим, например, ресурс, не имеющий ассоциированных данных, такой как "право использовать регистр ввода/вывода *n*". В принципе, если какая-то функция получает этот ресурс, но не освобождает его, то может возникнуть утечка этого ресурса. Чтобы PREfast мог выявить утечку ресурса, он должен быть в силах идентифицировать экземпляр ресурса, которым владеет или требует функция. Таким образом, для регистра ввода/вывода необходимо аннотировать "право использовать регистр ввода/вывода *n*" как ресурс.

В терминах функции, которая получает право использовать этот ресурс, *n* является параметром для нее, и необходимо создать объект, представляющий для PREfast "право использовать регистр ввода/вывода *n*". Но в анализируемой функции ни один объект не удерживает "право использовать регистр ввода/вывода *n*", поэтому во время эмуляции PREfast не может ничего перегрузить для хранения этой концепции.

В примере в листинге 23.16 показывается, как с помощью аннотации *\_drv\_acquiresResourceGlobal* отобразить определение конкретного экземпляра на имя класса. В этом примере идентификатор *IORRegister* является именем класса, а параметр *regnum* идентифицирует экземпляра ресурса.

#### Листинг 23.16. Пример аннотации для функции, получающей глобальный ресурс

```
_checkReturn
_drv_acquiresResourceGlobal(IORRegister, regnum)
NTSTATUS acquireIORRegister(int regnum);
```

Функция *acquireIORRegister* помещает имя в определенное закрытое пространство имен, недоступное для PREfast, где хранится право на использование ресурса *regnum* типа *IORRegister*. Когда освобождается право на использование регистра, идентификатор указывает, что это право больше не удерживается. Это служит той же самой цели, что и указатель на память, возвращаемый функцией *malloc*, не внося ничего в исходный код анализируемой функции.

Обратите внимание на то, что аннотация *\_drv\_acquiresResourceGlobal* не применяется к параметру *regnum* никаким образом. Вместо этого, *regnum* — или, вернее, его значение, эмулированное PREfast — является параметром для этой аннотации, точно так же, как и *size* в аннотации *\_bcnt(size)* является параметром для этой аннотации.

### Аннотации для критической области и спин-блокировки отмены

Следующие аннотации можно использовать для аннотирования функций, которые получают или освобождают критическую область или спин-блокировку отмены (cancel spin lock).

```
_drv_acquiresCriticalSection
_drv_releasesCriticalSection
```

```
_drv_acquiresCancelSpinLock
_drv_releasesCancelSpinLock
```

Определенные ресурсы, такие как критическая область и спин-блокировка отмены, не имеют программных имен — они просто существуют. Только один экземпляр таких ресурсов может существовать в любой данный момент. Для аннотирования этих ресурсов можно применить ту же самую концепцию, что применяется для аннотации `_drv_acquiresResourceGlobal`, как было описано в предыдущем разделе, но часть имени, указывающая конкретный экземпляр (т. е. второй параметр), задается неявно именем макроса. На практике, для этого типа ресурсов применяются макросы специального назначения.

Как частный случай для большинства драйверов, доступ к двум глобальным ресурсам — критической области и спин-блокировке отмены — осуществляется посредством контекста. Ни один из ресурсов не имеет ни имени, ни значения параметра. Функция `KeEnterCriticalSection` не имеет ни параметров, ни возвращаемого значения, она просто выполняет блокировку до успешного завершения.

Это можно было бы выразить с помощью более общей аннотации, но рекомендуемым подходом является использование аннотаций для критической области и спин-блокировки отмены. Эти аннотации также указывают, что при выполнении операции получения ресурса не может быть уже полученным, а при выполнении операции освобождения ресурс уже должен быть полученным. Для получения дополнительной информации см. разд. "Аннотации для удерживания и неудерживания ресурсов, иных, чем память" далее в этой главе.

В примере, приведенном в листинге 23.17, указывается, что функция получает критическую область. Данная аннотация является предпочтительным способом для выражения аннотации `_drv_acquiresResourceGlobal(CriticalRegion, "")`.

#### Листинг 23.17. Пример аннотации для функции, получающей критическую область

```
_drv_acquiresCriticalSection
VOID
KeEnterCriticalSection();
```

Для указания, что получение ресурса может завершиться неудачей, можно применять аннотации для условий, как объясняется в разд. "Аннотации для условий" ранее в этой главе. Для ресурсов иных, чем память, указание успеха или неудачи обычно отделено от самого ресурса. Например, в листинге 23.18 показано, что функция `KeTryToAcquireSpinLockAtDpcLevel` возвращает `TRUE` только после получения спин-блокировки.

#### Листинг 23.18. Пример аннотации для функции, получение ресурса которой может завершиться неудачей

```
NTKERNELAPI
BOOLEAN
_drv_valueIs (==0;==1)
FASTCALL
KeTryToAcquireSpinLockAtDpcLevel (
    _inout _drv_when(return==1,
        _drv_deref(_drv_acquiresResource(SpinLock)))
    PKSPIN_LOCK SpinLock
);
```

### **Аннотации для удерживания и неудерживания ресурсов, иных, чем память**

Для указания ресурсов, которые должны или не должны удерживаться функцией, можно использовать следующие аннотации:

```
_drv_mustHold( kind)
_drv_neverHold( kind)
_drv_mustHoldGlobal( kind, param)
_drv_neverHoldGlobal( kind, param)
_drv_mustHoldCriticalSection
_drv_neverHoldCriticalSection
_drv_mustHoldCancelSpinLock
_drv_neverHoldCancelSpinLock
```

Для некоторых функций требуется, чтобы при их вызове ресурс уже удерживался или наоборот — не удерживался. В качестве самого простого примера таких функций можно привести специальные функции для критических областей и спин-блокировки отмены, непосредственно перед получением или освобождением ресурса. Но несколько других функций не работают должным образом, если соответствующие ресурсы не были выделены или не выделены. Например, функция `ExAcquireResourceExclusiveLite` должна вызываться, когда ей уже была выделена критическая область.

### **Аннотации для выделенных функции ресурсов, иных, чем память**

Аннотации `_drv_mustHold(kind)` и `_drv_neverHold(kind)` можно применять ко всей функции, а не только к определенному параметру. Эти аннотации указывают, что при вызове функции должен быть уже выделен, по крайней мере, один ресурс `kind` или, наоборот, не выделено ни одного такого ресурса.

Например, функцию `IoCompleteRequest` нельзя вызывать, когда удерживается любая спин-блокировка. Это обстоятельство указывается с помощью аннотации `_drv_neverHold(SpinLock)`. Аннотации `_drv_mustHold(Memory)` и `_drv_neverHold(Memory)` указывают, что удерживаемый объект является объектом памяти, например, из функции `malloc` или `new`.

### **Аннотации для выделенного глобального ресурса, иного, чем память**

Аннотации `_drv_mustHoldGlobal(kind, param)` и `_drv_neverHoldGlobal(kind, param)` указывают, что глобальный ресурс должен удерживаться или не удерживаться. Параметры аннотаций `kind` и `param` имеют то же самое значение, как и такие же параметры в аннотациях для получения и освобождения глобальных ресурсов.

### **Аннотации для удерживания критической области или спин-блокировки отмены**

Следующие аннотации используются с критической областью и спин-блокировкой отмены:

```
_drv_mustHoldCriticalSection
_drv_neverHoldCriticalSection
_drv_mustHoldCancelSpinLock
_drv_neverHoldCancelSpinLock
```

Примеры использования этих аннотаций приводятся в листингах 23.19—23.21.

**Листинг 23.19. Пример аннотации для функции, которая должна удерживать критическую область**

```
_drv_mustHoldCriticalSection
BOOLEAN
ExAcquireResourceExclusiveLite(
    _in PERESOURCE Resource,
    _in BOOLEAN Wait
);
```

**Листинг 23.20. Пример аннотации для функции, которая никогда не должна удерживать спин-блокировку**

```
_drv_neverHold(SpinLock)
VOID
IoCompleteRequest(
    _in PIRP Irp,
    _in CCHAR PriorityBoost
);
```

Пример в листинге 23.21 содержит аннотации для предотвращения получения или освобождения ресурса более одного раза.

**Листинг 23.21. Пример аннотации для функций, которые не должны получать или освобождать ресурс более одного раза**

```
VOID
KeAcquireSpinLock(
    _inout _deref(_drv_acquiresResource(SpinLock))
        _drv_neverHold(SpinLock)
    PKSPIN_LOCK SpinLock,
    _out PKIRQL OldIrql
);
VOID
KeReleaseSpinLock(
    _inout _deref(_drv_releasesResource(SpinLock))
        _drv_mustHold(SpinLock)
    PKSPIN_LOCK SpinLock,
    _in KIRQL NewIrql
);
```

## Составные аннотации для ресурсов

Для функций, выделяющих ресурсы, можно применять следующие аннотации:

```
_drv_acquiresExclusiveResource(kind)
_drv_releasesExclusiveResource(kind)
_drv_acquiresExclusiveResourceGlobal(kind, param)
_drv_releasesExclusiveResourceGlobal(kind, param)
```

Эти составные аннотации применяются для функций выделения ресурсов, которые похожи на оберточные функции спин-блокировок в том отношении, что у ресурса может быть только один владелец:

- ◆ формы "acquires" аннотаций совмещают аннотации `_drv_neverHold` и `_drv_acquiresResource`;
- ◆ формы "releases" аннотаций совмещают аннотации `_drv_mustHold` и `_drv_releasesResource`.

В табл. 23.6 приводится краткое описание этих аннотаций.

**Таблица 23.6. Составные аннотации для ресурсов**

| Аннотация                                                      | Описание                                                                                                                   |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>_drv_acquiresExclusiveResource(kind)</code>              | Функция получает ресурс <code>kind</code> и не может уже удерживать этот ресурс                                            |
| <code>_drv_releasesExclusiveResource(kind)</code>              | Функция освобождает ресурс <code>kind</code> и должна уже иметь этот ресурс                                                |
| <code>_drv_acquiresExclusiveResourceGlobal(kind, param)</code> | Функция получает экземпляр <code>param</code> глобального ресурса <code>kind</code> и не может уже удерживать этот ресурс  |
| <code>_drv_releasesExclusiveResourceGlobal(kind, param)</code> | Функция освобождает экземпляр <code>param</code> глобального ресурса <code>kind</code> и должна уже удерживать этот ресурс |

В примере, приведенном в листинге 23.22, указывается, что функция получает ресурс `MySpinLock`, которого она еще не должна иметь.

#### Листинг 23.22. Пример составной аннотации для получения спин-блокировки

```
VOID
GetMySpinLock(
    _inout  _deref( _drv_acquiresExclusiveResource (MySpinLock) )
        PKSPIN_LOCK SpinLock,
    _out   PKIRQL OldIrql
);
```

## Аннотации класса типа функций

Драйверы постоянно определяют функции, вызываемые системой с помощью указателя на функцию, который драйвер передает системе. Примерами таких функций обратного вызова могут служить драйверные функции `AddDevice`, `StartIo` и `Cancel`. Многие из рассматриваемых в этой главе аннотаций применяются к таким функциям обратного вызова.

Должным образом аннотированные функции обратного вызова существенно облегчают для PREfast задачу проверки на правильность использования функции. Это можно сделать, объявив функции членами класса типа функции.

Большинство распространенных системных типов обратного вызова имеют класс функции. Для драйверов WDM системные классы типов функций определены в заголовочном файле `%wdk%\inc\ddk\Wdm.h`. PREfast может распознавать аннотации класса типа функции для драйверов WDM. Аннотации классов для драйверов KMDF определены в заголовочном файле `%wdk%\inc\wdf\kmdf\wdffroletypes.h`. Функциональность распознавания этих ролевых классов встроена в инструмент Static Driver Verifier и планируется для будущих версий PREfast.

С помощью объявлений `typedef` можно определять собственные классы типов функций, как описано в разд. "Аннотации к объявлениям `typedef` функций" ранее в этой главе.

## Аннотации для идентификации класса типа функции

Аннотация `_drv_functionClass(name)` указывает, что функция или объявление `typedef` функции является членом именованного класса функций, представленного посредством параметра `name`. Эта аннотация наиболее полезна, когда применяется как к функции, так и к типу указателя на функцию. Легче всего ее применять посредством объявлений функции `typedef`.

Аннотация `_drv_functionClass` распространяется на соответствующее объявление указателя функции `typedef` и заставляет PREfast проверять, чтобы при назначениях функции указателю на функцию и при отмене таких назначений применялся совпадающий класс функции.

PREFast выдает предупреждение в следующих случаях:

- ◆ функция не имеет класса функции;
- ◆ указателю на функцию, имеющему класс, назначается функция несовпадающего класса.

Чтобы удалить причину этого предупреждения, необходимо добавить в код соответствующее определение функции `typedef`. Необходимое объявление функции `typedef` обычно приводится в тексте сообщения об ошибке, выдаваемого PREFast.

Например, в листинге 23.23 аннотация `_drv_functionClass` указывает, что это объявление функции `typedef` для класса функции `DRIVER_INITIALIZE`. Имя `typedef` и имя класса находятся в разных пространствах имен. Указатель `PDRIVER_INITIALIZE` принадлежит к классу `DRIVER_INITIALIZE`, т. к. он наследует от объявления `typedef DRIVER_INITIALIZE`.

### Листинг 23.23. Пример аннотации для функции определенного класса типа функции

```
typedef _drv_functionClass(DRIVER_INITIALIZE)
NTSTATUS
DRIVER_INITIALIZE (
    _in struct _DRIVER_OBJECT *DriverObject,
    _in PUNICODE_STRING RegistryPath
);
typedef DRIVER_INITIALIZE *PDRIVER_INITIALIZE;
```

## Аннотации для проверки класса типа функции в условном выражении

Аннотации для некоторых функций действительны только тогда, когда функция вызывается (или не вызывается) из функций определенного типа, указанного аннотацией `_drv_functionClass`. Эту ситуацию можно аннотировать, добавив вызов функции `isFunctionClassS(name)` в условное выражение в аннотации `_drv_when`.

Эта функция определяет, относится ли функция к классу, идентифицируемому параметром `name`:

- ◆ если нет, то функция `isFunctionClassS` возвращает 0;
- ◆ если да, то 1.

Например, аннотация `isFunctionClassS("DRIVER_INITIALIZE")` определяет, является ли функция процедурой типа инициализации драйвера. Для получения информации о других функциях, которые можно использовать в условных выражениях, см. разд. "Специальные функции в условных выражениях" ранее в этой главе.

Примером особого случая, применимого только к унаследованным драйверам, является функция `IoCreateDevice`. Для унаследованных драйверов система разрешает вызов функции `IoCreateDevice` из функции `DRIVER_INITIALIZE` и не требует, чтобы эта функция явно отслеживала полученный объект устройства. Система помещает объект устройства в такое место, где функция `Unload` драйвера может найти его, но PREfast не может и поэтому выдает предупреждение.

В следующем примере показано применение аннотаций для предотвращения шума, вызываемого в особом случае применением PREfast для проверки унаследованных драйверов:

```
_drv_when(!isFunctionClass("DRIVER_INITIALIZE") && return==0,
          _deref(_drv_allocatesMem(DeviceObject)))
```

Выражение `return==0` является примером проверки на успешное выполнение.

## Аннотации для плавающей запятой

Для некоторых семейств процессоров, в особенности для семейства процессоров *x86*, использование плавающей запятой из кода режима ядра разрешено только функциям, которые сохраняют и восстанавливают состояние плавающей запятой. Обнаружить нарушения этого правила может быть трудной задачей, т. к. они вызывают только нерегулярные проблемы во время исполнения. С помощью соответствующих аннотаций PREfast может выявить использование плавающей запятой в коде режима ядра и предупредить об ошибке, если состояние плавающей запятой не защищено должным образом. Проверка на соблюдение правил для плавающей запятой выполняется только для кода режима ядра.

Указать действия функции с состоянием плавающей запятой можно с помощью аннотаций `_drv_floatSaved` и `_drv_floatRestored`. Эти аннотации уже применены к системным функциям `KeSaveFloatingPointState` и `KeRestoreFloatingPointState`, как и аннотации для получения и освобождения ресурсов с целью предотвращения утечки ресурсов. Подобные функции семейства `EngXXX` тоже аннотированы таким образом. Но аннотации также должны применяться и для оберточных функций для этих функций.

Когда PREfast обнаруживает кажущееся незащищенное использование плавающей запятой, то он выдает предупреждение. Если вся функция вызывается безопасным образом какой-либо функцией, то к ней можно применить аннотацию `_drv_floatUsed`, которая подавляет предупреждение и заставляет PREfast проверить правильность использования функции вызывающим клиентом. По мере необходимости можно добавлять дополнительные уровни аннотации `_drv_floatUsed`. PREfast автоматически применяет аннотацию `_drv_floatUsed`, когда или результат функции, или один из ее параметров являются типом с плавающей запятой, но может быть полезным применять аннотацию явным образом, в качестве документирования.

Например, в листинге 23.24 аннотация `_drv_floatSaved` указывает, что состояние плавающей запятой сохраняется в параметре `FloatSave` системной функции `KeSaveFloatingPointState`.

**Листинг 23.24. Пример аннотации, указывающей местонахождение состояния плавающей запятой**

```
NTSTATUS KeSaveFloatingPointState(
    _out _drv_deref(_drv_floatSaved)
        PKFLOATING_SAVE FloatSave
);
```

А в листинге 23.25 аннотация `_drv_floatUsed` подавляет предупреждения PREfast об использовании состояния плавающей запятой функцией `MyDoesFloatingPoint`. Эта аннотация также заставляет PREfast выполнять проверку, чтобы функция `MyDoesFloatingPoint` всегда вызывалась в безопасном контексте плавающей запятой.

**Листинг 23.25. Пример аннотаций для функции, использующей плавающую запятую**

```
_drv_floatUsed
VOID
MyDoesFloatingPoint(arguments);
```

## Аннотации для уровней IRQL

Все драйверы режима ядра при исполнении должны принимать во внимание уровни IRQL. Когда PREfast анализирует аннотированный код драйвера, он пытается предугадать диапазон значений уровней IRQL, в котором могла бы исполняться функция, и выявить любые несоответствия.

Аннотации для IRQL помогают PREfast делать более точные предположения о диапазоне уровней IRQL, в котором должна исполняться функция. Например, функцию можно аннотировать максимальным уровнем IRQL, на котором ее можно вызывать. Потом, если эта функция вызывается на более высоком уровне IRQL, PREfast выдает предупреждение об ошибке. Чем больше аннотаций применяется к драйверным функциям, тем лучше может PREfast делать такие выводы и тем лучше может обнаруживать ошибки.

Аннотации параметров уровней IRQL взаимодействуют друг с другом больше, чем другие аннотации, т. к. значения уровней IRQL устанавливаются, переустанавливаются, сохраняются и возобновляются в вызовах различных функций.

В табл. 23.7 приводится список аннотаций для указания требуемого уровня IRQL для функций и параметров функций.

**Таблица 23.7. Аннотации для уровней IRQL**

| Аннотация                             | Описание                                                                                                                                                                    |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_drv_maxIRQL(value)</code>      | Параметр <code>value</code> указывает максимальный уровень IRQL, на котором можно вызывать функцию                                                                          |
| <code>_drv_minIRQL(value)</code>      | Параметр <code>value</code> указывает минимальный уровень IRQL, на котором можно вызывать функцию                                                                           |
| <code>_drv_setsIRQL(value)</code>     | Функция возвращает уровень IRQL                                                                                                                                             |
| <code>_drv_requiresIRQL(value)</code> | Вход в функцию должен выполняться на уровне IRQL, указанном в параметре <code>value</code>                                                                                  |
| <code>_drv.raisesIRQL(value)</code>   | Функция возвращает управление на уровне IRQL, указанном в параметре <code>value</code> , но ее можно вызывать только для повышения, а не для понижения текущего уровня IRQL |
| <code>_drv_savesIRQL</code>           | Аннотированный параметр сохраняет текущий уровень IRQL для дальнейшего восстановления                                                                                       |
| <code>_drv_restoresIRQL</code>        | Аннотированный параметр содержит значение уровня IRQL из аннотации <code>_drv_savesIRQL</code> , который необходимо восстановить после возвращения управления функцией      |

Таблица 23.7 (окончание)

| Аннотация                                             | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| _drv_savesIRQLGlobal( <i>kind</i> , <i>param</i> )    | Текущий уровень IRQL сохраняется во внутренней области памяти PREfast, из которого его нужно восстановить. Эта аннотация применяется с функциями. Область памяти указывается параметром <i>kind</i> и уточняется параметром <i>param</i>                                                                                                                                                                                                                                                                                                           |
| _drv_restoresIRQLGlobal( <i>kind</i> , <i>param</i> ) | Уровень IRQL, сохраненный функцией с аннотацией _drv_savesIRQLGlobal, восстанавливается из внутренней области памяти PREfast                                                                                                                                                                                                                                                                                                                                                                                                                       |
| _drv_minFunctionIRQL( <i>value</i> )                  | В параметре <i>value</i> указывается минимальное значение уровня IRQL, к которому функция может понизить уровень IRQL                                                                                                                                                                                                                                                                                                                                                                                                                              |
| _drv_maxFunctionIRQL( <i>value</i> )                  | В параметре <i>value</i> указывается максимальное значение уровня IRQL, к которому функция может повысить уровень IRQL                                                                                                                                                                                                                                                                                                                                                                                                                             |
| _drv_sameIRQL                                         | Аннотированная функция получает и возвращает управление на одном и том же уровне IRQL. Функция может изменить уровень IRQL, но она должна восстановить его первоначальное значение перед возвращением управления                                                                                                                                                                                                                                                                                                                                   |
| _drv_isCancelIRQL                                     | Аннотированный параметр указывает уровень IRQL, передаваемый в вызове функции обратного вызова DRIVER_CANCEL. Эта аннотация указывает, что функция является утилитой, которая вызывается из процедур Cancel и которая завершает выполнение требований для функций DRIVER_CANCEL, включая освобождение спин-блокировки отмены.<br>Аннотация _drv_isCancelIRQL является составной аннотацией, состоящей из аннотации _drv_useCancelIRQL и нескольких других аннотаций, которые обеспечивают правильную работу функции обратного вызова DRIVER_CANCEL |
| _drv_useCancelIRQL                                    | Аннотированный параметр является значением IRQL, которое функция обратного вызова DRIVER_CANCEL должна восстановить.<br>При самостоятельном применении аннотации _drv_useCancelIRQL может быть полезной только в редких случаях, например, когда остальные требования, описываемые аннотацией _drv_isCancelIRQL, были уже удовлетворены каким-либо образом                                                                                                                                                                                         |

## Аннотации для указания максимального и минимального уровня IRQL

Аннотации \_drv\_maxIRQL(*value*) и \_drv\_minIRQL(*value*) просто указывают, что функцию нельзя вызывать на уровне IRQL, ниже или выше указанного в параметре *value*. Например, когда PREfast видит последовательность вызовов функций, которые не изменяют уровень IRQL, если он обнаруживает вызов на уровне, указанном в аннотации \_drv\_maxIRQL, который ниже, чем уровень, указанный в близлежащей аннотации \_drv\_minIRQL, то PREfast выдает предупреждение на втором вызове функции, который он обнаруживает. Ошибка в действительности может находиться в первом вызове функции, но предупреждение только указывает местонахождение второй части конфликта.

Если в аннотациях функции упоминается уровень IRQL, но явно не применяется аннотация `_drv_maxIRQL`, то PREfast неявно применяет аннотацию `_drv_maxIRQL(DISPATCH_LEVEL)`. За редкими исключениями, это обычно правильный уровень. Неявное применение этой аннотации по умолчанию помогает избежать загромождения кода аннотациями, а также делает исключения более заметными.

Аннотация `_drv_minIRQL(PASSIVE_LEVEL)` всегда применяется неявно, т. к. не может быть более низких уровней IRQL. Таким образом, соответствующего явного правила о минимальном уровне IRQL не существует. Очень немногие функции имеют верхнюю границу уровня IRQL, иную, чем `DISPATCH_LEVEL`, а нижнюю — иную, чем `PASSIVE_LEVEL`.

Некоторые функции вызываются в контексте, в котором вызываемая функция не может безопасно повысить уровень IRQL выше определенного максимума или, что случается чаще, не может безопасно понизить его ниже определенного минимума. Аннотации `_drv_maxFunctionIRQL` и `_drv_minFunctionIRQL` помогают PREfast обнаруживать случаи, где повышение или понижение может произойти непреднамеренно.

Например, для функций типа `DRIVER_STARTIO` применяются аннотации `_drv_minFunctionIRQL(DISPATCH_LEVEL)`. Это означает, что при исполнении такой функции, понижение уровня IRQL меньше уровня `DISPATCH_LEVEL` является ошибкой. Другие аннотации указывают, что вход в функцию и выход из нее должны выполняться на уровне `DISPATCH_LEVEL`.

## Аннотации для явного указания уровня IRQL

Аннотации `_drv_setsIRQL` и `_drv_requiresIRQL` помогают PREfast лучшим образом докладывать о противоречивостях, обнаруженных с аннотациями `_drv_maxIRQL` и `_drv_minIRQL`, т. к. тогда PREfast знает уровень IRQL.

## Аннотации для повышения и понижения уровня IRQL

Аннотация `_drv_raisesIRQL` похожа на аннотацию `_drv_setsIRQL`, но указывает, что функция должна использоваться только для повышения уровня IRQL и не должна применяться для понижения уровня IRQL, даже если синтаксис функции позволяет это. Примером функции, которая не должна применяться для понижения уровня IRQL, является функция `KeRaiseIrql`.

## Аннотации для сохранения и восстановления уровня IRQL

К аннотациям для сохранения и восстановления уровня IRQL относятся следующие аннотации:

```
_drv_savesIRQL
_drv_restoresIRQL
_drv_savesIRQLGlobal(kind, param)
_drv_restoresIRQL Global(kind, param)
```

Аннотации `_drv_savesIRQL` и `_drv_restoresIRQL` указывают, что текущий уровень IRQL, известный точно или только приблизительно, сохраняется в аннотированном параметре или восстанавливается из него соответственно. Предполагается, что ассоциированные с аннотациями `_drv_savesIRQL` и `_drv_restoresIRQL` переменные являются какими-либо целочисленными типами, т. е. любым целочисленным типом, допустимым компилятором. Всегда, когда возможно, PREfast пытается работать с такими значениями, как с целыми числами.

Некоторые функции сохраняют и восстанавливают уровень IRQL неявным образом. Например, функция `ExAcquireFastMutex` сохраняет значение уровня IRQL в непрозрачной области

памяти, ассоциированной с объектом быстрого мьютекса, который указывается в первом параметре функции. Сохраненное значение уровня IRQL восстанавливается соответствующей функцией `ExReleaseFastMutex` для этого объекта быстрого мьютекса. Эти действия можно указать явным образом с помощью аннотаций `_drv_savesIRQLGlobal(kind, param)` и `_drv_restoresIRQLGlobal(kind, param)`. Параметры `kind` и `param` указывают, где сохраняется значение уровня IRQL, подобно параметрам для аннотаций ресурсов, рассмотренных в разд. "Аннотации для ресурсов иных, чем память" ранее в этой главе. При условии согласующихся аннотаций для сохранения и восстановления значения уровня IRQL не требуется точно указывать область хранения значения.

## **Аннотации для удерживания постоянного уровня IRQL**

Определяемые пользователем функции, которые изменяют уровень IRQL, необходимо аннотировать либо с помощью аннотации `_drv_sameIRQL`, либо одной из других аннотаций для уровня IRQL, чтобы указывать, что ожидается изменение в уровне IRQL. При отсутствии аннотаций, указывающих любые конечные изменения уровня IRQL, PREfast выдает предупреждение для любой функции, выход из которой не осуществляется на том же самом уровне IRQL, на котором был осуществлен вход в нее. Если изменение в уровне IRQL является преднамеренным, то необходимо применить соответствующую аннотацию, чтобы подавить такое сообщение. Если же изменение уровня IRQL не является преднамеренным, то нужно найти в коде источник ошибки и исправить ее.

Применение аннотации `_drv_sameIRQL` дает знать другим программистам, что автор-разработчик считал такое поведение правильным. Например, аннотация `_drv_sameIRQL` применяется почти со всеми системными функциями обратного вызова, т. к. от них ожидается осуществлять выход на том же самом уровне IRQL, на каком был осуществлен вход в них. Исключения помечаются в соответствии с исполняемыми ими действиями.

## **Аннотации для сохранения и восстановления уровня IRQL для процедур отмены ввода/вывода**

Аннотация `_drv_useCancelIRQL` указывает, что параметр, к которому она относится, является значением уровня IRQL, который необходимо восстановить функцией обратного вызова `DRIVER_CANCEL`. Эта аннотация указывает, что функция является утилитой, которая вызывается из процедур `Cancel` и которая завершает выполнение требований для функций `DRIVER_CANCEL` (т. е. она выполняет обязательства контракта для вызывающего клиента).

Например, сервисная функция `MyCompleteCurrent` вызывается из разных участков кода, чтобы реализовать функциональность отмены. Одним из параметров этой функции является значение уровня IRQL, который она должна восстановить. Аннотации указывают, что функция должна отвечать требованиям функции `Cancel`, как показано в следующем примере:

```
VOID
MyCompleteCurrent(
    _in PDEVICE_EXTENSION Extension,
    _in_opt PKSYNCHRONIZE_ROUTINE SynchRoutine,
    _in _drv_in(_drv_useCancelIRQL) KIRQL IrqlForRelease,
);

```

Обратите внимание, что аннотация `_drv_useCancelIRQL` является аннотацией низкого уровня. Для многих применений аннотация `_drv_isCancelIRQL` будет лучшим выбором.

## Примеры аннотаций IRQL

Максимальный уровень IRQL, на котором можно получить быстрый мьютекс, — уровень APC\_LEVEL. Операция получения быстрого мьютекса повышает уровень IRQL до APC\_LEVEL. При освобождении быстрого мьютекса вызывающий клиент все еще должен находиться на уровне IRQL, а первоначальный уровень IRQL восстанавливается. Пример применения аннотаций для осуществления этих правил приведен в листинге 23.26.

### Листинг 23.26. Пример аннотаций для соблюдения максимального уровня IRQL

```
_drv_maxIRQL(APC_LEVEL)
_drv_setsIRQL(APC_LEVEL)
VOID
ExAcquireFastMutex(
    _inout _drv_out(_drv_savesIRQL
                    _drv_acquiresResource(FastMutex))
    PFAST_MUTEX FastMutex
);

_drv_requiresIRQL(APC_LEVEL)
VOID
ExReleaseFastMutex(
    _inout _drv_in(_drv_restoresIRQL
                    _drv_releasesResource(FastMutex))
    PFAST_MUTEX FastMutex
);
```

В листинге 23.27 приведен пример аннотаций для замены уровня по умолчанию \_drv\_maxIRQL(DISPATCH\_LEVEL) для функции KeRaiseIrql и указания, что эта функция может только повышать уровень IRQL.

### Листинг 23.27. Пример аннотаций для подмены максимального уровня IRQL по умолчанию

```
_drv_maxIRQL(HIGH_LEVEL)
VOID
KeRaiseIrql(
    _in _drv_in(_drv_raisesIRQL) KIRQL NewIrql,
    _out _drv_out_deref(_drv_savesIRQL) PKIRQL OldIrql
);
```

В примере, приведенном в листинге 23.28, обратите внимание на ассоциацию сохраненного и восстановленного уровней IRQL с параметром LockHandle, который не ассоциируется непосредственно с сохраненным значением уровня IRQL.

### Листинг 23.28. Аннотации для сохранения и восстановления уровня IRQL

```
_drv_maxIRQL(DISPATCH_LEVEL)
_drv_savesIRQLGlobal(QueuedSpinLock, LockHandle)
_drv_setsIRQL(DISPATCH_LEVEL)
VOID
```

```

FASTCALL
KeAcquireInStackQueuedSpinLock (
    _in PKSPIN_LOCK SpinLock,
    _in _drv_in_deref(_drv_acquiresExclusiveResource(QueuedSpinLock))
    PKLOCK_QUEUE_HANDLE LockHandle
);
_drv_restoresIRQLGlobal(QueuedSpinLock, LockHandle)
_drv_requiresIRQL(DISPATCH_LEVEL)
VOID
FASTCALL
KeReleaseInStackQueuedSpinLock (
    _in _drv_in_deref(_drv_releasesExclusiveResource(QueuedSpinLock))
    PKLOCK_QUEUE_HANDLE LockHandle
);

```

Требования по минимальному и максимальному значениям изменения уровня IRQL обычно применяются к функциям обратного вызова. В примере в листинге 23.29 аннотации указывают, что функция `theAddDevice` вообще не может повышать уровень IRQL.

#### Листинг 23.29. Пример аннотаций для предотвращения повышения уровня IRQL функцией

```

typedef
_drv_maxFunctionIRQL(0)
_drv_sameIRQL
_drv_clearDoInit(yes)
_drv_functionClass(DRIVER_ADD_DEVICE)
NTSTATUS
DRIVER_ADD_DEVICE (
    _in struct _DRIVER_OBJECT *DriverObject,
    _in struct _DEVICE_OBJECT *PhysicalDeviceObject
);
typedef DRIVER_ADD_DEVICE *PDRIVER_ADD_DEVICE;

```

#### Полезные советы по применению аннотаций IRQL

Далее приводится несколько полезных советов по применению аннотаций IRQL с функциями.

- ◆ Представьте PREfast как можно больше информации об уровнях IRQL, снабжая функции соответствующими аннотациями. Такая дополнительная информация будет полезной для PREfast при последующей проверке как вызывающей, так и вызываемой функции. В некоторых случаях вставка аннотаций позволяет подавить ложноположительные сообщения.
- ◆ Если в функции нет никаких явных аннотаций для уровня IRQL, это, скорее всего, служебная функция, которую можно вызывать на любом уровне IRQL; поэтому отсутствие явных аннотаций для уровня IRQL является правильным способом аннотирования данной функции.
- ◆ При аннотировании функции под IRQL примите во внимание возможную будущую реализацию функции, а не только текущую.

Например, текущая реализация функции может позволять ей работать правильно на более высоких уровнях IRQL, чем предполагалось разработчиком. Хотя аннотирование

функции на основе фактических возможностей кода может выглядеть привлекательно, разработчик должен принимать во внимание будущие требования, такие как, например, необходимость понизить максимальный уровень IRQL для какого-либо будущего улучшения или находящегося в стадии воплощения системного требования. Аннотации должны отражать намерения разработчика функции, а не фактическую ее реализацию.

## Аннотация ***DO\_DEVICE\_INITIALIZING***

Аннотация `_drv_clearDoInit` указывает, что функция должна очистить бит `DO_DEVICE_INITIALIZING` в слове `Flags` объекта устройства. Вызов функции с этой аннотацией удовлетворяет это требование для вызывающего клиента.

Эту аннотацию следует применять в условном контексте, когда функция возвращает статус успешного завершения, за исключением случаев, когда аннотация применяется к объявлению функции `typedef`. Для примера см. разд. "Примеры аннотаций IRQL" ранее в этой главе.

## Аннотации для операндов с взаимоблокировкой

Большое семейство функций принимают в качестве одного из своих параметров адрес переменной, к которой необходимо обращаться с помощью блокировочных инструкций процессора. Это атомарные инструкции для чтения кэша. Неправильное использование таких операндов может вызывать труднообнаруживаемые неочевидные ошибки.

Параметр функции можно идентифицировать как взаимоблокируемый операнд с помощью аннотации `_drv_interlocked`. Системные функции уже имеют аннотации для взаимоблокируемых операндов.

PREfast предполагает, что если к переменной осуществляется доступ посредством любой блокировочной функции, то разработчик планирует разделение переменной потоками, которые могут исполняться на отдельных процессорах. Таким образом, любая попытка обратиться к этой переменной или модифицировать ее без применения блокировочной информации может быть осуществлена только в кэше локального процессора, а такой код может быть потенциально неправильным. Присутствие переменных в локальном стековом фрейме, использующемся в качестве блокируемого операнда, является как очень необычным, так и часто опасным, и, как правило, является признаком некорректного использования функции.

В следующем коде приводится пример аннотации для функции `InterlockedExchange`. Эта аннотация указывает, что к параметру `target` всегда необходимо обращаться посредством блокировочной операции:

```
LONG  
InterlockedExchange(  
    _inout _drv_in(_drv_interlocked) PLONG Target,  
    _in LONG Value  
) ;
```

## Примеры аннотированных системных функций

В данном разделе приводятся примеры применения аннотаций к часто используемым системным функциям.

В листинге 23.30 функция возвращает значение посредством параметра, который необходимо проверить. Также приводятся примеры других аннотаций.

**Листинг 23.30. Пример аннотации для функции IoGetDmaAdapter**

```
PDMA_ADAPTER
_drv_maxIRQL(DISPATCH_LEVEL)
IoGetDmaAdapter(
    _in PDEVICE_OBJECT PhysicalDeviceObject,
    _in PDEVICE_DESCRIPTION DeviceDescription,
    _checkReturn
    _deref_inout PULONG NumberOfMapRegisters
);

```

Функция `IoCreateDevice` обычно очень проста в применении, за исключением того, что драйвер должен соответствующим образом удалить созданный объект устройства, что обычно делается вызовом функции `IoAttachDeviceToDeviceStack`. Но когда наследуемый драйвер вызывает функцию `IoCreateDevice` из функции драйвера `DRIVER_INITIALIZE` или `DRIVER_DISPATCH`, то объект устройства помещается в область памяти, где его можно впоследствии найти, обычно при выгрузке драйвера. Проблема заключается в том, что PREfast не знает об этой области памяти.

В листинге 23.32 приводится пример аннотаций для предотвращения ложноположительных результатов анализов PREfast в такой ситуации. Объект устройства похож на память в том отношении, что к нему можно применять совмещение имен, поэтому в примере используется аннотация для памяти.

**Листинг 23.31. Пример аннотации для IoCreateDevice**

```
_drv_maxIRQL(APC_LEVEL)
NTSTATUS
_drv_valueIs(<0;==0)
IoCreateDevice(
    _in PDRIVER_OBJECT DriverObject,
    _in ULONG DeviceExtensionSize,
    _in_opt PUNICODE_STRING DeviceName,
    _in DEVICE_TYPE DeviceType,
    _in ULONG DeviceCharacteristics,
    _in BOOLEAN Exclusive,
    _out _drv_out(_drv_when(return<0, _null)
                  _drv_when(return==0, _notnull
                            _drv_when(!inFunctionClass$("DRIVER_INITIALIZE")
                                      && !inFunctionClass$("DRIVER_DISPATCH"),
                                      _acquiresMemory(Memory)))) )
    PDEVICE_OBJECT *DeviceObject
);

```

Для симметрии с функцией `IoCreateDevice` в листинге 23.32 показан пример использования аннотаций в функции `IoAttachDeviceToDeviceStack`. Обратите внимание на то, что в этом примере совмещение имен для объекта устройства применяется только в случае успешного завершения функции. Правильно написанный код должен проверить возвращаемое значение функции `IoAttachDeviceToDeviceStack` и в случае неуспешного ее завершения обычно должен вызвать функцию `IoDeleteDevice`. Эти аннотации заставляют PREfast следовать этим правилам.

**Листинг 23.32. Пример аннотаций для функции IoAttachDeviceToDeviceStack**

```
PDEVICE_OBJECT
_drv_maxIRQL(2)
_drv_valueIs(==0; !=0)
_checkReturn
IoAttachDeviceToDeviceStack(
    _in PDEVICE_OBJECT SourceDevice,
    _in _drv_in(_drv_mustHold(Memory)
        _drv_when(return!=0, _drv_aliasesMem))
    PDEVICE_OBJECT TargetDevice
);

```

В листинге 23.33 приведен пример использования аннотации для функции EngSaveFloatingPointState, которая сохраняет текущее состояние плавающей точки ядра. Хотя с виду прототип функции кажется простым, на уровне контракта семантика этой функции достаточно сложная и с легкостью поддается некорректному применению. Соответственно, аннотации для указания корректного использования функции также достаточно сложные.

В этом примере используется аннотация `_drv_arg`, чтобы переместить более сложные аннотации из списка параметров в блок аннотаций перед началом прототипа функции. Простые аннотации, такие как `_in`, оставлены в их обычном месте.

**Листинг 23.33. Пример аннотаций для функции EngSaveFloatingPointState**

```
// EngSaveFloatingPointState
_checkReturn
_drv_arg(*pBuffer, // 1
    _drv_neverHold(EngFloatState) // 2
    _drv_notPointer() // 3
    _drv_when(pBuffer==0 || cjBufferSize==0,
        _drv_ret(_drv_valueIs(>=0))) // 4
    _drv_when(pBuffer!=0 && cjBufferSize!=0,
        _drv_ret(_drv_valueIs(==0; ==1)) // 5
    _drv_when(return==1, _drv_ret(_drv_floatSaved)
        _drv_arg(pBuffer, _bcount_opt(cjBufferSize)
            _deref _drv_acquiresResource(EngFloatState))
    )
)
ULONG EngSaveFloatingPointState(
    _out_opt VOID *pBuffer, // 6
    _in ULONG cjBufferSize // 6
);

```

Далее приводится объяснение пронумерованных комментариев фрагментов листинга 23.33:

1. Конечное значение всегда должно проверяться или использоваться.
2. При вызове функции не должно удерживаться состояние плавающей запятой.
3. Указатель буфера должен указывать непосредственно в память.

4. Если любой из параметров равняется нулю, то функция возвращает положительное число (т. е. размер буфера, требуемого для сохранения состояния плавающей запятой) или ноль (т. е. процессор не поддерживает на аппаратном уровне плавающую запятую).
5. Если значение обоих параметров не равно нулю, то функция возвращает значение типа `BOOLEAN`. В случае успеха функция сохраняет состояние плавающей запятой, заполняет буфер и получает состояние плавающей запятой из буфера.
6. Обычные аннотации `_in` и `_out` по-прежнему применяются.

## Составление и отладка аннотаций

Составление многих аннотаций просто и очевидно, но для некоторых требуется приложение определенных усилий, чтобы получить желаемый результат. PREfast пытается диагностировать каждую ошибку, которую он находит в аннотациях, но он не может проверить и доложить обо всех возможных ошибках. Поэтому всегда будет хорошей идеей составить небольшой контрольный пример, на котором можно подтвердить, что аннотации работают, как и предполагалось.

Хороший контрольный пример должен отлавливать все ожидаемые ошибки, но не должен докладывать о правильных применениях аннотаций. Простые аннотации, такие как `_in`, не извлекают пользы из контрольных примеров, но для аннотаций, связанных с размерами (в частности, аннотации, в которых используется модификатор `_part`), контрольные примеры часто оказываются полезными, т. к. написание контрольного примера часто заставляет разработчика задуматься о нетипичных ситуациях.

## Образцы контрольных примеров для аннотаций

В листинге 23.34 показан набор очень простых контрольных примеров для функции `myfun`, которая принимает три параметра: `mode`, `p1` и `p2`. Значение параметра `mode` определяет действительные значения для параметров `p1` и `p2`, поэтому контрольные примеры используют условную аннотацию `_drv_when` для указания, каким образом принуждать корректное применение функции `myfun`.

### Листинг 23.34. Образец контрольных примеров аннотаций для функции

```
// Аннотируем прототип функции.
_drv_when(mode==1, _drv_arg(p1, _null))
_drv_when(mode==2, _drv_arg(p2, _null))
_drv_when(mode <= 0 || mode > 2, _drv_reportError("bad mode value"))
void myfun(_in int mode,
           _in struct s *p1,
           _in struct s *p2);
```

Начиная с первой аннотации, прототип устанавливает следующие требования:

- ◆ если значение `mode` равно 1, то значение `p1` должно быть `NULL`;
- ◆ если значение `mode` равно 2, то значение `p2` должно быть `NULL`;
- ◆ если значение параметра `mode` меньше либо равно 0 или больше 2, то выдается сообщение об ошибке "bad mode value" ("неверное значение параметра mode").

Контрольные примеры в листинге 23.35 вызывают функцию правильным и неправильным образом. PREfast реагирует на неправильные вызовы, выдавая предупреждения.

#### Листинг 23.35. Образец кода для проверки контрольных примеров аннотаций

```
// Правильное использование
void dummy1(_in struct s *a, _in struct s *b)
{
    myfun(0, a, b);
}

// Неправильное использование: следует ожидать предупреждения
void dummy2(_in struct s *a, _in struct s *b)
{
    myfun(1, a, b);
}

// Неправильное использование: следует ожидать предупреждения
void dummy3(_in struct s *a, _in struct s *b)
{
    myfun(2, a, b);
}

// Правильное использование
void dummy4(_in struct s *a, _in struct s *b)
{
    myfun(1, NULL, b);
}

// Правильное использование
void dummy5(_in struct s *a, _in struct s *b)
{
    myfun(2, a, NULL);
}

// Неправильное использование: следует ожидать предупреждения
void dummy6(_in struct s *a, _in struct s *b)
{
    myfun(14, a, b);
}
```

## Полезные советы для написания контрольных примеров для аннотаций

При написании контрольных примеров для аннотаций примите к рассмотрению следующие обстоятельства.

- ◆ Обычно код контрольного примера не должен выполнять никаких операций. Часто достаточно только выполнить вызов функции внутри фиктивной функции.
- ◆ Если требуются указатели на структуры, то часто достаточно, чтобы фиктивная функция принимала эти указатели в качестве параметров, к которым применены соответствующие аннотации `_in`, `_out` или другие необходимые для тестирования аннотации.
- ◆ Контрольные примеры для аннотаций не нужно ни компоновать, ни исполнять. Все, что необходимо, — это скомпилировать их с помощью PREfast.
- ◆ При написании нескольких контрольных примеров будет хорошей идеей поместить каждый из них в отдельную фиктивную функцию. PREfast старается минимизировать шум,

подавляя повторяющиеся в функции ошибки. Часто при попытках обнаружить другие способы активирования ошибки логика подавления повторяющихся ошибок подавляет дополнительные случаи.

- ◆ Для контрольных примеров, как правило, лучше подходят функции, возвращающие `void`, т. к. от них не требуется удовлетворять требованиям компилятора, чтобы функция возвращала значение.
- ◆ Не забудьте присвоить каждой функции индивидуальное имя.
- ◆ Помните, что аннотации независимы друг от друга. Обычно нет надобности в проверке на неожиданное взаимодействие несвязанных аннотаций.

## Оптимальные методики работы с PREfast

Далее приводится сводка оптимальных методик для работы с PREfast и использования аннотаций.

### Оптимальные методики для PREfast

Примите во внимание следующие общие оптимальные методики для интегрирования PREfast в свой цикл разработки.

- ◆ Установите политики для интегрирования PREfast в свои практические разработки.

Примите к сведению следующие идеи по интегрированию PREfast в ваши проекты вашей команды разработчиков:

- начинайте использовать PREfast сразу же по получению безошибочной компиляции каждого исходного файла;
- установите политику для вашей команды разработчиков относительно того, какие предупреждения PREfast необходимо исправлять, какие можно игнорировать, а какие разработчик может по своему усмотрению или исправлять, или игнорировать;
- установите политику и соглашения для протоколирования причин, по которым конкретные предупреждения, такие как ложноположительные предупреждения, не нужно исправлять.

- ◆ Используйте фильтры, чтобы анализировать результаты PREfast более эффективно.

Для решения, какие предупреждения необходимо скрывать, попробуйте следовать этим рекомендациям:

- как минимум, исправьте все предупреждения, которые проходят через фильтр `drivers-all`. Это особенно серьезные предупреждения с низким уровнем шума об ошибках, которые могут повлиять на безопасность системы;
- фильтруйте результаты PREfast, скрывая сообщения, для которых команда разработчиков считает, что риск, связанный с сообщением, является приемлемо низким, при высоком уровне шума, связанным с сообщением, а также когда цикл отправки продукта позволяет исправлять только наиболее критические ошибки.

◆ **Придерживайтесь вариантов кодирования, которые помогают уменьшить уровень шума PREfast.**

Далее приводится несколько рекомендаций по общим практикам кодирования и комментирования.

- Число ложноположительных предупреждений, вызванных неадекватными методами написания кода, можно уменьшить внесением в код незначительных изменений.
- Вместо применения вставляемого кода ассемблера, используйте сервисные функции, предоставляемые более новыми компиляторами.

Если избежать применения вставляемого кода ассемблера невозможно, поместите его в функцию `_inline`, чтобы PREfast мог анализировать код более эффективно. Применяйте аннотации для этих функций.

- Всегда, когда возможно, инициализируйте переменные при их объявлении.
- Идентифицируйте делаемые в коде предположения с помощью комментариев, и сделайте эти предположения явными с помощью формальных утверждений (assertions).
- Применяйте скобки и другие средства синтаксиса для принуждения порядка выполнения арифметических действий, вместо того, чтобы полагаться на правила предшествования языка С, в случаях, когда эти правила не полностью интуитивны, т. е. в случаях, когда PREfast выдает предупреждения.
- Используйте макрос `NT_SUCCESS` вместо явных проверок на `STATUS_SUCCESS`.
- Начните использовать аннотации, чтобы предоставить PREfast более конкретную информацию о разрабатываемом исходном коде.
- В добавление к PREfast продолжайте использовать другие инструменты тестирования и проверки достоверности драйверов, такие как Static Driver Verifier и Driver Verifier.

## Оптимальные методики для использования аннотаций

Далее приводятся оптимальные методы для применения аннотаций.

◆ **Используйте аннотации способом, имеющим смысл для данного проекта.**

"Правильным" подходом к использованию аннотаций является тот, который работает лучшим образом для вашего проекта.

- Для некоторых проектов может иметь смысл использование возможностей PREfast в самой полной мере: активно исследуйте каждую функцию в программе и применяйте соответствующие аннотации и другие рекомендуемые изменения.
- Это требует больших усилий, но помогает обеспечить, чтобы каждая ошибка, которую PREfast способен обнаружить, была обнаружена и идентифицирована. При правильном исполнении это также минимизирует ложноположительные предупреждения.
- В случае нехватки времени и ограниченных ресурсов может быть разумным применить реактивный подход, заключающийся в анализе существующего кода с помощью PREfast и применении аннотации, необходимых для подавления ложноположительных предупреждений.

- Это не обязательно позволит обнаружить все ошибки, которые могли бы быть выявлены применением проактивного подхода, но таким образом можно начать использовать PREFast.

Вместо того чтобы откладывать применение PREFast до того момента, "когда есть время делать это правильно", лучше применить реактивный подход, или даже не вставлять аннотаций вообще, а просто игнорировать предупреждения. Обычно время, сэкономленное благодаря тому, что PREFast обнаружит даже несколько ошибок, достаточно для того, чтобы компенсировать усилия, вложенные в применение аннотаций, по сравнению со временем, требуемым для отладки этих проблем обычными методами. Но не стоит останавливаться на реактивном подходе, а с течением времени постепенно переходить к проактивной модели, и тогда польза от еще большего уровня проактивности должна стать очевидной.

#### ◆ **Применяйте аннотации для описания успешного вызова функции.**

Аннотации должны отражать успешный вызов функции. PREFast должен улавливать плохо сконструированные или неудачные вызовы, а не только вызовы, ведущие к фатальным системным сбоям. Аннотации должны отражать предназначение функции, выраженные ее интерфейсом, а не фактическую реализацию функции.

Распространенным примером могут служить необязательные параметры. Бывает, что для проверки значения параметра на NULL пишется большой объем кода. Но является ли значение NULL параметра в действительности необязательным? Например:

- если функция игнорирует значение NULL параметра и продолжает работу, делая что-то полезное, тогда параметр является необязательным;
- если при вызове со значением параметра NULL функция возвращает ошибку, то тогда параметр не является необязательным. Вместо этого функция защищается от некорректного применения.

Если PREFast может сообщитьзывающему клиенту, что возможное значение параметра NULL вызовет неудачное завершение функции, то он может выявить эту потенциальную ошибку, вместо того, чтобы она всплыла во время исполнения при неясных и трудно поддающихся тестированию обстоятельствах. Реализация функции таким образом, чтобы она могла защищаться против некорректных параметров, является хорошей практикой, но не делает параметр необязательным в смысле этого слова.

#### ◆ **Обдумайте возможное эволюционирование функции.**

При аннотировании функции важно принять во внимание возможную будущую реализацию функции, а не только текущую. Аннотация должна отражать намерения разработчика функции, а не ее фактическую текущую реализацию.

Например, текущая реализация функции может позволить ей работать правильно с иными значениями параметров, чем предусматривалось разработчиком. Хотя аннотирование функции на основе фактических возможностей кода может выглядеть привлекательно, разработчик должен принимать во внимание будущие требования, например, необходимость реализовать какое-либо ограничение для поддержки какого-нибудь будущего улучшения или находящегося в стадии воплощения системного требования.

#### ◆ **Устраняйте конфликты между реализацией функции и ее документацией.**

Аннотации могут выявить конфликты двух типов между реализацией функции и ее документацией:

- код и документация не соответствуют друг другу — одно или другое необходимо исправить;
- в документации описывается фактический, но не применяемый аспект функции, т. е. "некорректный" вызов функции завершится удачей.

В таком случае разработчик функции должен решить, нужно ли исправить документацию или же принять меры для воплощения задокументированного поведения функции, с тем, чтобы функция использовалась, как и предполагалось.

◆ **Применяйте флаги компилятора /w4, /wx и /w64.**

Компилятор также выявляет определенное количество возможных ошибок, не все из которых обнаруживаются PREfast. Хорошей практикой будет использование следующих флагов компилятора:

- /w4 — предупреждения выдаются на уровне 4;
- /wx — предупреждения показываются как ошибки;
- /w64 — выдавать предупреждения по вопросам совместимости с 64-битными системами.

Сведите к минимуму область видимости аннотаций предупреждений `#pragma`, используемых для подавления ложноположительных предупреждений компилятора.

◆ **Для кода драйверов применяйте только аннотации, описанные в этой главе.**

В зависимости от версии используемых инструментов анализа, аннотации реализуются разными способами. Документацию или комментарии в заголовочных файлах, содержащие слово `_declspec`, и аннотации, в которых используются квадратные скобки, подобные применяемым в языке C#, можно игнорировать для целей использования PREfast. Для кода драйверов официально поддерживаются только аннотации, рассматриваемые в этой главе.

### Примечание

Аннотации можно также предоставлять посредством файла модели. По разным причинам многие системные файлы Microsoft аннотированы в файле модели, но не в исходном коде. Аннотации, предоставляемые файлом модели, ничем не лучше аннотаций, описанных в этой главе и, кроме того, их труднее использовать. Поэтому использование файла модели не задокументировано и не рекомендуется для новых аннотаций.

В настоящее время некоторые аннотации реализованы лишь частично. Также аннотации могут быть достаточными для подавления ложных предупреждений в PREfast, но не для выполнения дополнительных проверок, которых следовало бы ожидать из названия аннотации. Бывают еще аннотации, которые являются просто структуризованными комментариями. Эти аннотации планируется использовать в будущих версиях PREfast.

## Пример: аннотированный заголовочный файл Osrusbfx2.h

В листинге 23.36 приведен пример заголовочного файла, содержащего аннотации PREfast `_drv_requiresIRQL`, `_in` и `_out`.

В этом файле также приведены аннотации для типа роли функций обратного вызова KMDF, такие как `EVT_WDF_DRIVER_DEVICE_ADD`, которые может интерпретировать инструмент Static Driver Verifier. Подробную информацию см. в главе 24.

**Листинг 23.36. Аннотированный заголовочный файл Osrusbf2.h**

```
#pragma warning(disable:4200) // Структура/объединение без имени
#pragma warning(disable:4201) // Структура/объединение без имени
#pragma warning(disable:4214) // Типы битовых полей, иные, чем int
#include <initguid.h>
#include <ntddk.h>
#include "usbdi.h"
#include "usbdlib.h"
#include "public.h"
#include "driverspecs.h"
#pragma warning(default:4200)
#pragma warning(default:4201)
#pragma warning(default:4214)
#include <wdf.h>
#include <wdfusc.h>
#define NTSTRSAFE_LIB
#include <ntstrsafe.h>
#include "trace.h"
#ifndef _PRIVATE_H
#define _PRIVATE_H
#define POOL_TAC (ULONG) 'FRSO'
#define _DRIVER_NAME_ "OSRUSBFX2:"
#define TEST_BOARD_TRANSFER_BUFFER_SIZE (64*1024)
#define DEVICE_DESC_LENCTH 256
//
// Определения команд поставщика, поддерживаемых устройством
//
#define USBFX2LK_READ_7SECMENT_DISPLAY 0xD4
#define USBFX2LK_READ_SWITCHES 0xD6
#define USBFX2LK_READ_BARCRAPH_DISPLAY 0xD7
#define USBFX2LK_SET_BARCRAPH_DISPLAY 0xD8
#define USBFX2LK_IS_HICH_SPEED 0xD9
#define USBFX2LK_REENUMERATE 0xDA
#define USBFX2LK_SET_7SECMENT_DISPLAY 0xDB
//
// Определения возможностей устройства, которые
// можно устанавливать и сбрасывать
//
#define USBFX2LK_FEATURE_EPSTALL 0x00
#define USBFX2LK_FEATURE_WAKE 0x01
//
// Порядок конечных точек в описателе интерфейса
//
#define INTERRUPT_IN_ENDPOINT_INDEX 0
#define BULK_OUT_ENDPOINT_INDEX 1
#define BULK_IN_ENDPOINT_INDEX 2
//
// Структура, предоставляющая информацию об экземпляре,
// ассоциированном с этим конкретным устройством.
//
typedef struct _DEVICE_CONTEXT {
    WDFUSBDEVICE UsbDevice;
    WDFUSBINTERFACE UsbInterface;
```

```
WDFUSBPIPE           BulkReadPipe;
WDFUSBPIPE           BulkWritePipe;
WDFUSBPIPE          InterruptPipe;
UCHAR                CurrentSwitchState;
WDFQUEUE             InterruptMsgQueue;
} DEVICE_CONTEXT,    *PDEVICE_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, GetDeviceContext)
extern ULONG DebugLevel;
DRIVER_INITIALIZE DriverEntry;
EVT_WDF_OBJECT_CONTEXT_CLEANUP OsrFxEvtDriverContextCleanup;
EVT_WDF_DRIVER_DEVICE_ADD OsrFxEvtDeviceAdd;
EVT_WDF_DEVICE_PREPARE_HARDWARE OsrFxEvtDevicePrepareHardware;
EVT_WDF_IO_QUEUE_IO_READ OsrFxEvtIoRead;
EVT_WDF_IO_QUEUE_IO_WRITE OsrFxEvtIoWrite;
EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL OsrFxEvtIoDeviceControl;
EVT_WDF_REQUEST_COMPLETION_ROUTINE EvtRequestReadCompletionRoutine;
EVT_WDF_REQUEST_COMPLETION_ROUTINE EvtRequestWriteCompletionRoutine;
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
ResetPipe(
    _in WDFUSBPIPE Pipe
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
ResetDevice(
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
SelectInterfaces(
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
AbortPipes(
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
ReenumerateDevice(
    _in PDEVICE_CONTEXT DevContext
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
GetBarGraphState(
    _in PDEVICE_CONTEXT DevContext,
    _out PBAR_CGRAPH_STATE BarGraphState
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
SetBarGraphState(
    _in PDEVICE_CONTEXT DevContext,
    _in PBAR_CGRAPH_STATE BarGraphState
);
```

```
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
GetSevenSegmentState(
    _in PDEVICE_CONTEXT DevContext,
    _out PUCHAR SevenSegment
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
SetSevenSegmentState(
    _in PDEVICE_CONTEXT DevContext,
    _in PUCHAR SevenSegment
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
GetSwitchState(
    _in PDEVICE_CONTEXT DevContext,
    _in PSWITCH_STATE SwitchState
);
_drv_requiresIRQL(DISPATCH_LEVEL)
VOID
OsrUsbIoctlGetInterruptMessage (
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
OsrFxSetPowerPolicy(
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
NTSTATUS
OsrFxConfigContReaderForInterruptEndPoint(
    _in PDEVICE_CONTEXT DeviceContext
);
_drv_requiresIRQL(PASSIVE_LEVEL)
VOID
OsrFxEvtUsbInterruptPipeReadComplete(
    _in WDFUSBPIPE Pipe,
    _in WDFMEMORY Buffer,
    _in size_t NumBytesTransferred,
    _in WDFCONTEXT Context
);
_drv_requiresIRQL(PASSIVE_LEVEL)
BOOLEAN
OsrFxEvtUsbInterruptReadersFailed(
    _in WDFUSBPIPE Pipe,
    _in NTSTATUS Status,
    _in USBD_STATUS UsbdStatus
);
EVT_WDF_IO_QUEUE_IO_STOP OsrFxEvtIoStop;
EVT_WDF_DEVICE_D0_ENTRY OsrFxEvtDeviceD0Entry;
EVT_WDF_DEVICE_D0_EXIT OsrFxEvtDeviceD0Exit;
_drv_requiresIRQL(PASSIVE_LEVEL)
BOOLEAN
```

```
OsrFxReadFdoRegistryKeyValue(
    _in PWDDEVICE_INIT DeviceInit,
    _in PWCHAR           Name,
    _out PULONG          Value
);
_drv_requiresIRQL(PASSIVE_LEVEL)
VOID
OsrFxEnumerateChildren(
    _in WDFDEVICE Device
);
_drv_requiresIRQL(PASSIVE_LEVEL)
PCHAR
DbgDevicePowerString(
    _in WDF_POWER_DEVICE_STATE Type
);
#endif
```

## ГЛАВА 24

# Инструмент Static Driver Verifier

Инструмент для статического анализа Static Driver Verifier (в дальнейшем просто SDV) предназначен для автоматической проверки во время компиляции написанного на языке С кода драйверов Windows с целью выявления нарушений правил KMDF и WDM. В этой главе описывается принцип работы SDV и представляются основы использования этого инструмента для верификации драйверов KMDF.

| Ресурсы, необходимые для данной главы                            | Расположение                                                                                            |
|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <strong>Инструменты и файлы</strong>                             |                                                                                                         |
| Инструмент SDV в наборе разработчика WDK                         | %wdk%\tools\sdv (Windows Server Longhorn Beta 3 или более поздняя версия Windows)                       |
| Wdf.h header file                                                | %wdk%\inc\wdf\kmdf                                                                                      |
| Dispatch_routines.h                                              | %wdk%\tools\sdv\osmodel\wdf                                                                             |
| <strong>Образцы драйверов</strong>                               |                                                                                                         |
| Osrusbfx2                                                        | %wdk%\src\kmdf\osrusbfx2                                                                                |
| Дефектные драйверы KMDF                                          | %wdk%\tools\sdv\samples\fail_drivers\kmdf                                                               |
| Дефектные драйверы WDM                                           | %wdk%\tools\sdv\samples\fail_drivers\wdm                                                                |
| <strong>Документация WDK</strong>                                |                                                                                                         |
| Документация для Static Driver Verifier                          | <a href="http://go.microsoft.com/fwlink/?LinkId=80084">http://go.microsoft.com/fwlink/?LinkId=80084</a> |
| <strong>Прочее</strong>                                          |                                                                                                         |
| Раздел <strong>Static Driver Verifier</strong> на Web-сайте WHDC | <a href="http://go.microsoft.com/fwlink/?LinkId=80082">http://go.microsoft.com/fwlink/?LinkId=80082</a> |
| Microsoft Research: SLAM                                         | <a href="http://research.microsoft.com/slam">http://research.microsoft.com/slam</a>                     |

## Введение в SDV

Обнаружение и анализ ошибок в драйверах Windows могут быть сложной задачей даже при всех предоставляемых WDF преимуществах. Драйверы режима ядра работают в асинхронном и массово реентерабельном режиме, при этом каждый драйвер должен следовать множеству сложных правил, чтобы правильно использовать сотни DDI-функций, предоставляемых

мых WDM и KMDF. Кроме этого, задача тестирования драйверов усложняется следующими обстоятельствами:

- ◆ ошибки во взаимодействии между драйвером и Windows недоступны для непосредственного обозрения;
- ◆ драйвер может работать должным образом большую часть времени, но в исключительных ситуациях в нем могут происходить неочевидные ошибки;
- ◆ если нарушение драйвером явных правил работы вызывает его нестабильную работу или, еще хуже, сбой системы, обнаружение причины ошибки может быть очень трудной задачей.

SDV расширяет возможности наблюдения за ошибками при тестировании драйверов, отслеживая соблюдение драйвером правил KMDF и WDM, которые определяют использование драйверами устройств соответствующих функций интерфейса DDI.

Инструмент для статической верификации SDV применяется в конце цикла разработки для проверки исполняемых драйверов, работа над которыми подходит к фазе тестирования. SDV выполняет более глубокий анализ, чем PREfast, и часто находит ошибки после того, как все выявленные PREfast ошибки были исправлены. Для работы SDV требуется Windows XP или более поздние версии Windows. SDV поддерживает свободные и проверочные среды для x86 и x64.

### Внимание!

Чтобы при разработке драйверов KMDF воспользоваться возможностями SDV в полной мере, необходимо пользоваться версией набора разработчика WDK, поставляемой с Windows Server Longhorn Beta 3 или более поздней версией Windows. Для получения самой последней информации о применении SDV для разработки драйверов Windows см. раздел **Static Driver Verifier** на Web-сайте WHDC по адресу <http://go.microsoft.com/fwlink/?LinkId=80082>.

Лучше всего SDV работает с драйверами небольшого объема, которые имеют меньше чем 50 тысяч строк кода. SDV может проверять драйверы, имеющие следующие характеристики:

- ◆ драйверы и библиотеки, написанные на языке C, в исходных файлах с расширением C;
- ◆ драйверы устройств KMDF или WDM (т. е. функциональные драйверы, драйверы фильтров и драйверы шины) и драйверы фильтров файловой системы.

### **SDV и подразделение Microsoft Research**

Научно-исследовательское подразделение корпорации Microsoft, называющееся Microsoft Research (MSR), затратило несколько лет работы на создание движка для автоматической проверки правильного использования интерфейсов к внешним библиотекам программами, написанными на языке C. Результатом работы над этим проектом (называющимся SLAM в MSR) является мощный движок анализа, который был внедрен в SDV.

Команда центра MSR описывает процесс работы над движком в разделе **Providing a Template for Tech Transfer** на Web-сайте по адресу <http://research.microsoft.com/slam> и <http://research.microsoft.com/displayArticle.aspx?id=1338>.

## Принцип работы SDV

SDV исследует ветви исполнения кода драйвера устройства, символически исполняя исходный код драйвера. Он помещает драйвер в агрессивную среду и систематически проверяет

все ветви исполнения кода, выискивая нарушения правил использования KMDF или WDM. Для символического исполнения делается лишь незначительное число предположений о состоянии операционной системы Windows или начального состояния драйвера, поэтому SDV может проверять код под нагрузкой в ситуациях, которые трудно поддаются проверке посредством обычных методов тестирования. Для проверки драйвера SDV применяет следующий трехэтапный процесс.

1. Выполняет компиляцию, компоновку и сборку драйвера, применяя стандартную утилиту Build.
2. Сканирует исходный код драйвера в поисках точек входа.

Для драйверов KMDF точками входа являются все функции обратного вызова драйвера, а для драйверов WDM — процедуры в таблице диспетчеризации (dispatch table). SDV собирает список точек входа в файл Sdv-map.h, которым он потом руководствуется в процессе верификации драйвера.

Для получения подробной информации о файле Sdv-map.h см. разд. "Как отсканировать исходный код для создания файла Sdv-map.h" далее в этой главе.

3. Проводит подготовку к верификации, после чего проверяет, что драйвер следует правилам, указанным для текущего сеанса верификации.

Для подробной информации по этому вопросу см. разд. "А что внутри? Принцип работы движка верификации SDV" и "Как подготовить файлы и выбрать правила для SDV" далее в этой главе.

Для целей выполнения верификации SDV предоставляет свою собственную модель операционной системы. Движок верификации всесторонне анализирует ветви исполнения кода и пытается доказать, что драйвер нарушает правила, некорректно взаимодействуя с моделью операционной системы, предоставленной SDV.

Когда SDV удается доказать, что драйвер нарушил какое-либо правило, он объявляет ошибку и дает результатам верификации оценку Fail (Неудача). Если же SDV не может доказать никаких нарушений, то результатам верификациидается оценка Pass (Успех).

Движок верификации SDV проверяет по одному правилу за раз, до тех пор, пока не будет выполнена проверка на соответствие всем правилам. В процессе верификации SDV выводит в окно консоли сообщения о текущем статусе, а также сообщения об ошибках, как описано в разд. "Просмотр отчетов SDV" далее в этой главе.

## Правила SDV

Правила SDV составляются на языке, который называется Specification Language for Interface Checking<sup>1</sup> (для C) или SLIC. Когда SDV подготавливает драйвер для верификации, он использует конструкции языка SLIC, чтобы оснастить драйвер дополнительными операторами на языке C, которые описывают требуемое взаимодействие между драйвером и моделью операционной системы. Данную оснастку можно выполнить только с помощью SDV; вручную это сделать нельзя. Результаты применяются с копией исходного кода драйвера, а сам драйвер не модифицируется.

---

<sup>1</sup> Язык спецификаций для проверки интерфейса. — Пер.

Правило SDV может содержать один или несколько из следующих элементов.

- ◆ **Переменные состояния (state variables).** Эти элементы улавливают состояния, релевантные к процессу верификации кода драйвера.
- ◆ **Действия на входе и выходе.** Эти элементы являются входами в процедуры драйвера или модели операционной системы SDV и выходами из них. Действие указывает точки в коде (в драйвере или в модели операционной системы), в которые необходимо вставить соответствующий инструментарий.
- ◆ **Инструментарий.** Эти элементы состоят из условных операторов, присвоений значений переменным состояния и конструкции `abort`, указывающие запрещенное состояние, которое никогда не должно установиться в правильно написанном, работающим должным образом драйвере.
- ◆ **Ограничители.** Эти элементы представляют собой неявные условия, накладываемые на аргумент указателя процедуры действия. Для ссылки на аргумент применяется его положение: `$1` для первого аргумента, `$2` для второго и т. д., а для возвращаемого значения — `$return`. Аргумент должен указывать на тот же самый объект, на который указывает указатель "ограничитель" в определенной точке наблюдения в драйвере или в модели операционной системы SDV.

Ограничитель можно указать одним из следующих двух способов:

- ◆ конструкцией `with guard` — явным образом ссылается на переменную-ограничитель, а также процедуру, которая предоставляет свой вход в качестве единственной точки наблюдения;
- ◆ конструкцией `watch` — использует действие входа или выхода, дополненное позицией аргумента (`$1, $2, ..., $return`) для неявной ссылки на ограничитель. Набор наблюдаемых точек для ограничителя состоит из точек вызова процедуры действия `watch` в порядке их следования в коде драйвера. Во время верификации правило применяется в цикле для набора наблюдаемых точек — по одной итерации для каждой наблюдаемой точки.

Например, правило `RequestCompleted.slic` SDV для инфраструктуры KMDF проверяет, чтобы каждый запрос, поставленный в очередь ввода/вывода по умолчанию, был завершен драйвером (с некоторыми исключениями), как указывается инфраструктурой. Для правила `RequestCompleted.slic` применяется следующая конструкция:

```
state{
    enum {INIT, OK} t = INIT;
    enum {INIT1, REQPRESENTED} s = INIT1;
} with guard (sdv_main, hrequest)
```

В данном примере переменные состояния `t` и `s` могут принимать два возможных значения и инициализируются значениями `INIT` и `INIT1` соответственно.

А следующая конструкция говорит SDV, что процедура `sdv_main` является единственной наблюдаемой точкой для переменной указателя `hrequest`:

```
} with guard (sdv_main, hrequest)
```

Здесь:

- ◆ процедура `sdv_main` создана SDV в качестве обертки для драйвера;
- ◆ значением переменной указателя `hrequest` SDV является указатель на объект `WDFREQUEST`.

Среди прочих, правило RequestCompleted.slic содержит следующие действия входа:

```
fun_WDF_IO_QUEUE_IO_READ.entry[guard $2] {...}
sdv_WdfRequestComplete.entry [guard $1] {...}
```

Эти действия означают, что любой вызов процедуры обратного вызова *EvtIoRead* драйвера или метода *WdfRequestComplete* будет оснащен инструментарием для проверки его второго или, соответственно, первого параметра со значением наблюдаемой переменной *hrequest*.

Рассмотрим второй пример правила WDM SDV — правило *SpinLoc.slic*, которое проверяет, чтобы все спин-блокировки получались и освобождались попарно. Для этого правила применяется следующая конструкция:

```
watch sdv_KeAcquireSpinLock.exit.$1;
```

Это правило говорит SDV, что набор наблюдаемых точек содержит все выполняемые драйвером вызовы метода *KeAcquireSpinLock*. Если SDV обнаруживает в исходном коде драйвера два вызова метода *KeAcquireSpinLock*, он сохраняет ссылки на эти два вызова в файле *Slamwatchpoints.txt* и исполняет два отдельных сеанса верификации, по одному для каждого вызова.

## Как SDV применяет правила к коду драйвера

При запуске SDV ему указывается набор правил, на соблюдение которых необходимо проверить код драйвера. В процессе верификации SDV исследует все ветви кода драйвера и пытается доказать, что драйвер нарушает заданные правила. Если ему не удается доказать ни одного нарушения, то он сообщает, что драйвер удовлетворяет требованиям правил и успешно прошел верификацию. Если драйвер использует библиотеки, то если библиотеки были обработаны с помощью SDV перед выполнением верификации, SDV полностью анализирует каждую ветвь исполнения кода, включая участки в библиотеках.

Для верификации заданных правил SDV моделирует неблагоприятную среду драйвера, в которой может произойти несколько самых худших сценариев, например, постоянно завершающиеся неудачей вызовы операционной системы. SDV систематически проверяет все возможные ветви исполнения кода драйвера, выискивая нарушения заданных правил, определяющих должное взаимодействие между драйвером и инфраструктурой и между драйвером и ядром операционной системы. Например, правила драйверов KMDF для отмены запросов задают правила использования функций интерфейса DDI, принимающих участие в отмене запроса, который был поставлен в стандартную очередь ввода/вывода драйвера.

Определенные правила SDV являются предусловиями для других правил. Иными словами, если правило предусловия A представляет результат Pass или Fail, то этот результат определяет применимость к драйверу правила B. Например, если результатом правила предусловия FDODriver является Pass, то тогда правило *FDOPowerPolicyOwnerAPI* применимо к драйверу, но если результатом является Fail, то — нет. Таким образом, в последнем случае следует игнорировать все результаты проверки для драйвера, связанные с этим правилом.

Так как SDV выполняет проверку на серьезные ошибки в "закоулочных" ветвях исполнения кода, вероятность проверки которых даже при всестороннем обычном тестировании очень низкая, в зависимости от размера драйвера проверка по всем правилам может занять много времени и значительный объем физической памяти.

Чтобы получить наибольшую отдачу от правил SDV для драйверов KMDF, следует объявлять функции обратного вызова с соответствующими типами ролей функций, перечислен-

ными в заголовочном файле Dispatch\_routines.h (и в этой главе) и определенными в заголовочных файлах WDF. Чтобы предоставить драйверу доступ к типам ролей функций, необходимо включить заголовочный файл Wdf.h.

Для получения дополнительной информации об использовании правил SDV для драйверов см. разд. "Как аннотировать исходный код драйверов KMDF для SDV" далее в этой главе. А подробности о специфичных правилах для драйверов KMDF см. в разд. "Правила KMDF для SDV" далее в этой главе.

Для получения более подробной информации на эту тему см. отчет "Thorough Static Analysis of Device Drivers" (Всесторонний анализ драйверов устройств) конференции EuroSys по адресу <http://go.microsoft.com/fwlink/?LinkId=80612>.

## А что внутри?

### Принцип работы движка верификации SDV

Когда SDV выполняет проверку драйвера, он объединяет исходный код драйвера со своей моделью операционной системы, получая, таким образом, своеобразный "бутерброд" из кода:

- ◆ верхний слой этого "бутерброда" представляет собой управляющую оснастку, которая предоставляет сценарии вызовов функций обратного вызова KMDF или рабочих процедур драйверов WDM;
- ◆ средний слой — исходный код драйвера;
- ◆ а нижний слой содержит заглушки DDI вместо фактических функций WDF и WDM интерфейса DDI.

SDV вводит этот комбинированный код на языке C в верификационную машину SLAM. SLAM оснащает этот код правилами, выбранными при запуске процесса верификации, а затем выполняет всестороннюю верификацию этого кода.

Обратите внимание на то, что SLAM работает с копией драйвера; оригинальный исходный код драйвера не трогается.

Рассмотрим, например, следующее гипотетическое правило: функцию MyFunctionA нельзя вызывать после функции MyFunctionB. На языке SLIC это правило можно записать следующим образом:

```
state {  
    enum {False, True} WasMyFunctionBCalled = False;  
}  
MyFunctionB.entry {  
    WasMyFunctionBCalled = True;  
}  
MyFunctionA.entry {  
    if ( WasMyFunctionBCalled ) abort  
        "MyFunctionA is called after MyFunctionB"; }  
}
```

Обратите внимание в этом правиле на фрагменты кода на С между парами фигурных скобок. Когда SLAM оснащает драйвер правилами, он вставляет код на языке C для правила SDV в соответствующие места в исходном коде драйвера. В данном примере переменная WasMyFunctionBCalled инициализируется значением False. После этого SLAM вставляет следующий оператор присваивания в начало функции MyFunctionB:

```
WasMyFunctionBCalled = True;
```

SLAM также вставляет следующий оператор присваивания в начало функции `MyFunctionA`:

```
if (WasMyFunctionBCalled) SLIC_ABORT ("MyFunctionA is called  
after MyFunctionB");
```

После этого SLAM выполняет верификацию, итерируя через четыре этапа: абстракцию, поиск, проверку достоверности и усовершенствование.

## Абстракция

Сначала комбинированный и оснащенный правилами код приводится ("абстрагируется") к коду булевых абстракций. Код булевых абстракций похож на оригинальный код на языке С в области вызовов процедур и потока управления, но отличается от него в области данных. Он работает только с булевыми переменными, представляющими предикаты (т. е. булевые выражения), извлеченные из оригинального кода на языке С.

Первоначально, множество предикатов пустое, что означает, что условия в условных выражениях являются недетерминистическими, т. е. SLAM предполагает, что каждая ветвь любого условного оператора разрешена в любом состоянии. Таким образом, SLAM выполняет грубую избыточную аппроксимацию множества поведений драйвера способом, который гарантирует нахождение пути к `SLIC_ABORT`.

## Поиск и проверка достоверности

Полученная булева абстракция сначала очень тривиальна: в ней присутствуют только недетерминистические условия, и нет никаких присваиваний. Впоследствии абстракция усовершенствуется присваиваниям значений булевым переменным и выражением некоторых условий булевыми переменными. Имея эту абстракцию, SLAM выполняет на ней исчерпывающий поиск `SLIC_ABORT`. Сначала выполняется поиск в ширину, посредством вычисления в формате фиксированной запятой бинарной диаграммы решений, которая представляет семантику булевой абстракции.

В случае первоначальной избыточно аппроксимированной абстракции, путь ошибки к `SLIC_ABORT` может быть выявлен без больших проблем, если только первоначальная программа не является тривиально правильной, например, драйвер не вызывает функцию `MyFunctionA` вообще.

На этапе проверки достоверности SLAM символически эмулирует этот путь ошибки на оригинальном коде на языке С, принимая во внимание инструментарий, добавленный в драйвер.

Рассмотрим сначала случай, когда драйвер неправильно вызывает функцию `MyFunctionB` перед функцией `MyFunctionA` (т. е. драйвер нарушает правило). Рассмотрим для примера следующий код:

```
MyFunctionB(...); MyFunctionA(...);
```

Путь от входа в программу на языке С до вызова `SLIC_ABORT` сначала проходит через оператор инициализации `WasMyFunctionBCalled=False`, а потом через присваивание `WasMyFunctionBCalled=True`, которое было вставлено в начале тела функции `MyFunctionB`. Наконец, путь проходит через ветвь "истина" условного оператора `if (WasMyFunctionBCalled)`, который был вставлен в начале тела функции `MyFunctionA`.

Несомненно, этот путь вполне возможен; поэтому SLAM докладывает о нем, как об ошибке в драйвере. На этом проверка для этого случая завершена.

Теперь рассмотрим другой случай, когда драйвер не вызывает функцию MyFunctionB перед функцией MyFunctionA. Например, допустим, что драйвер настолько простой, что он вызывает только функцию MyFunctionA и никогда не вызывает функцию MyFunctionB. В этом случае путь от точки входа в программу на С до вызова SLIC\_ABORT опять проходит через инициализацию WasMyFunctionBCalled=False, а потом через ветвь "истина" условного оператора if (WasMyFunctionBCalled), который был вставлен в начале тела функции MyFunctionA. Но этот путь не содержит оператора присваивания WasMyFunctionBCalled=True, который был вставлен в начале тела функции MyFunctionB, т. к. драйвер не вызывает эту функцию. Очевидно, что данный путь невозможен, что SLAM определяет посредством следующего логического противоречия:

```
WasMyFunctionBCalled == False && WasMyFunctionBCalled == True
```

Когда SLAM обнаруживает это противоречие, он также обнаруживает релевантный предикат WasMyFunctionBCalled, который связан с противоречием. В данном случае SLAM продолжает проверку, т. к. предположительно дефектный путь в булевой абстракции кажется невозможным, или, по крайней мере, недостижимым, в соответствующем коде на языке С.

## Усовершенствование

Обнаружение логического противоречия в первоначальном эмулировании показывает, что WasMyFunctionBCalled является предикатом в противоречии. SLAM добавляет новые предикаты в первоначально пустое множество предикатов, после чего опять приводит исходный код на языке С к булевой абстракции. Данная булева абстракция более уточненная и тонкая, чем первоначальная, т. к. она работает с большим множеством предикатов.

Так как сейчас переменная WasMyFunctionBCalled находится во множестве предикатов, вторая булева абстракция также имеет все релевантные операторы, работающие с этой переменной, а именно оператор инициализации {WasMyFunctionBCalled = False;}. Теперь булева абстракция достаточно уточненная, чтобы в результате ее полного исследования поисковым алгоритмом не было обнаружено ни одного пути к SLIC\_ABORT. В самом деле, поиск не может следовать ветви "истинно" условия if (WasMyFunctionBCalled), вставленного в функцию MyFunctionA, т. к. сейчас во внимание принимается то обстоятельство, что переменная WasMyFunctionBCalled имеет значение False. Потом поиск докладывает, что булева абстракция не содержит никаких ошибок, чем подразумевается, что в данном примере оригинальный код драйвера следует правилу должным образом.

Для получения более глубокого объяснения процесса верификации см. литературу по SLAM по адресу <http://research.microsoft.com/slam>.

(*Влад Левин (Vlad Levin)*, член команды разработчиков Static Analysis Tools for Drivers, Microsoft.)

## Как аннотировать исходный код драйверов KMDF для SDV

В исходный код драйвера можно поместить аннотации, описывающие типы ролей функций обратного вызова KMDF для SDV, что значительно повысит возможность SDV анализировать код и обнаруживать в нем ошибки. Типы ролей предоставляют SDV информацию о предполагаемом использовании функции, что позволяет SDV определить лучшим образом, существует ли определенная ошибка.

Типы ролей функций для SDV делают известными индивидуальные роли, определенные для функций обратного вызова в базовой драйверной модели WDF, или для рабочих функций в базовой драйверной модели WDM. Эти типы ролей обычно либо заменяют, либо дополняют обычные объявления этих функций.

Чтобы позволить анализ драйвера KMDF инструментом SDV, в исходный код драйвера необходимо добавить объявления типов ролей, как изложено в этом разделе. Эти объявления типов ролей позволяют SDV сканировать драйвер, как описано в разд. "Как отсканировать исходный код для создания файла *Sdv-map.h*" далее в этой главе.

Для получения подробной информации по аннотированию драйверов KMDF или WDM объявлениями типов ролей см. разд. "Static Driver Verifier" в документации набора разработчика WDK.

## **Объявления типов ролей функций для драйверов KMDF**

Заголовочный файл Wdf.h содержит другие заголовочные файлы, в которых определено несколько типов ролей для функций обратного вызова драйверов, такие как, например, EVT\_WDF\_DRIVER\_DEVICE\_ADD, EVT\_WDF\_DRIVER\_UNLOAD и EVT\_WDF\_DEVICE\_FILE\_CREATE. Полный список типов ролей приводится в разд. "Типы ролей функций обратного вызова KMDF для SDV" в конце этой главы.

Чтобы улучшить возможности SDV, каждая функция обратного вызова драйвера KMDF должна быть объявлена в одном из заголовочных файлов драйвера, указывая соответствующий тип роли. Например, следующее объявление показывает тип роли для функции обратного вызова MyFileCreate драйвера:

```
EVT_WDF_DEVICE_FILE_CREATE MyFileCreate;
```

Это объявление, чисто на языке C, — все, что необходимо компилятору C, чтобы скомпилировать исходный код драйвера. Ни для компилятора C, ни для SDV не требуется традиционное подробное объявление функции обратного вызова.

Если вы предпочитаете традиционные подробные объявления, то объявление типа роли для SDV следует помещать непосредственно перед традиционным подробным объявлением, как показано в следующем примере:

```
EVT_WDF_DEVICE_FILE_CREATE MyFileCreate;
VOID MyFileCreate (
    IN WDFDEVICE Device,
    IN WDFREQUEST Request,
    IN WDFFILEOBJECT FileObject
);
```

## **Пример: использование типов ролей функций в образцах драйверов**

В листинге 24.1 приведены типы ролей для функций обратного типа образца драйвера KMDF Fail\_driver6, который находится в папке %wdk%\tools\sdv\samples\fail\_drivers\kmdf\fail\_driver6\driver. Объявления связанных функций находятся в файле FailDriver6.c.

**Листинг 24.1. Типы ролей SDV для функций обратного вызова образца драйвера Fail\_driver6**

```
#include <NTDDK.h>
#include <wdf.h>
#include "fail_library6.h"
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
EVT_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd;
EVT_WDF_IO_QUEUE_IO_READ EvtIoRead;
EVT_WDF_IO_QUEUE_IO_WRITE EvtIoWrite;
EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL EvtIoDeviceControl;
```

Тип роли функции EVT\_WDF\_DRIVER\_DEVICE\_ADD ассоциирован с функцией обратного вызова EvtDriverDeviceAdd драйвера. Так как типы ролей находятся в заголовочном файле Fail\_Driver6.h, добавлять объявления типов ролей в файл Fail\_Driver6.c нет необходимости.

В листинге 24.2 приведены типы ролей для функций обратного типа, объявленные в заголовочном файле для образца драйвера KMDF Osrusbf2, который находится в папке %wdk%\src\kmdf\osrusbf2.

**Листинг 24.2. Типы ролей SDV для функций обратного вызова образца драйвера Osrusbf2**

```
#include <wdf.h>
#include <wdfuscusb.h>
. . . // Код опущен для краткости.
typedef struct _DEVICE_CONTEXT {
    WDFUSBDEVICE          UsbDevice;
    WDFUSBINTERFACE        UsbInterface;
    WDFUSBPIPE             BulkReadPipe;
    WDFUSBPIPE             BulkWritePipe;
    WDFUSBPIPE             InterruptPipe;
    UCHAR                 CurrentSwitchState;
    WDFQUEUE               InterruptMsgQueue;
}
DEVICE_CONTEXT, *PDEVICE_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, GetDeviceContext)
extern ULONG DebugLevel;
DRIVER_INITIALIZE DriverEntry;
EVT_WDF_OBJECT_CONTEXT_CLEANUP OsrfxEvtDriverContextCleanup;
EVT_WDF_DRIVER_DEVICE_ADD OsrfxEvtDeviceAdd;
EVT_WDF_DEVICE_PREPARE_HARDWARE OsrfxEvtDevicePrepareHardware;
EVT_WDF_IO_QUEUE_IO_READ OsrfxEvtIoRead;
EVT_WDF_IO_QUEUE_IO_WRITE OsrfxEvtIoWrite;
EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL OsrfxEvtIoDeviceControl;
EVT_WDF_REQUEST_COMPLETION_ROUTINE EvtRequestReadCompletionRoutine;
EVT_WDF_REQUEST_COMPLETION_ROUTINE;
EvtRequestWriteCompletionRoutine;
. . .
```

Полный аннотированный листинг исходного кода образца драйвера Osrusbf2 приводится в главе 23.

## Использование SDV

Для применения SDV в процессе разработки драйверов необходимо подготовить файлы, выбрать правила, которые нужно проверить, после чего запустить SDV для выполнения верификации.

Проверять можно только один драйвер за раз. SDV исполняется в каталоге, содержащем файлы с исходным кодом и make-файлы драйвера. Если дерево сборки состоит из нескольких каталогов исходного кода драйвера, то необходимо запускать SDV отдельно в каждом каталоге.

### Как подготовить файлы и выбрать правила для SDV

Прежде чем выполнять проверку драйвера с помощью SDV, необходимо подготовить файлы и выбрать правила для верификации.

Для получения подробной информации по этому предмету см. раздел **Preparing to Run Static Driver Verifier** (Подготовка к использованию инструмента Static Driver Verifier) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80606>.

#### Подготовка к использованию SDV

1. Очистите папку sources драйвера.

Это не обязательно делать при первой проверке драйвера с помощью SDV, но обязательно для всех последующих проверок. Подробная информация по этому вопросу приводится в разд. "Очистка каталога sources драйвера" далее в этой главе.

2. Обработайте библиотеки драйвера.

Чтобы включить библиотеки в процесс верификации драйвера, необходимо обработать файлы исходного кода библиотеки, выполнив команду `staticdsv /lib` в каталоге sources библиотеки.

Этот шаг не является обязательным. Но если его не выполнить, то SDV не сможет выполнить в полной мере верификацию ветвлений кода в библиотеку. Обработка библиотек позволяет получить более аккуратные результаты верификации.

3. Просканируйте исходный код драйвера.

Эту операцию необходимо выполнить при первой проверке драйвера и после каждой модификации, которая изменяет точки входа драйвера.

Для получения подробной информации по этому вопросу см. разд. "Как отсканировать исходный код для создания файла Sdv-map.h" далее в этой главе.

4. В файле настроек SDV (`Sdv-default.xml`) установите временные пределы (`timeout`) и пределы виртуальной памяти (`spaceout`), применяемые SDV при верификации.

В данном файле также указывается число потоков, которые SDV может использовать для исполнения параллельных верификаций. По умолчанию число потоков равно 0, и автоматически устанавливается равным числу логических процессоров. Обратите внимание на то, что верификация с помощью SDV потребляет большое количество ресурсов, поэтому задание числа потоков выше, чем число логических потоков, не улучшит общую производительность.

Файл настроек хранится в папке %wdk%\tools\sdv\data\<WDM | KMDF>. Этот файл можно скопировать в любую папку sources драйвера, где его впоследствии можно редактировать нужным образом. Для получения дополнительной информации о файле настроек см. разд. "Static Driver Verifier Options File" в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80087>.

## 5. Определите правила, которые SDV должен верифицировать.

Перед запуском SDV определитесь, какое правило, или правила, необходимо верифицировать для драйвера. Проверка всех правил для большого драйвера может занять несколько часов.

Дополнительную информацию о правилах KMDF для SDV см. в разд. "Правила KMDF для SDV" далее в этой главе. Для получения дополнительной информации о правилах WDM для SDV см. разд. "Static Driver Verifier" в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80084>.

## Полезная информация

Для верификации нескольких драйверов с помощью SDV создайте командный файл, содержащий последовательность команд SDV.

## Очистка каталога sources драйвера

Перед верификацией драйвера и после его модификации нужно очистить папку sources драйвера, чтобы удалить файлы, созданные SDV в предыдущем сеансе верификации. Также, в случае модификации какой-либо библиотеки, используемой драйвером, необходимо повторно обработать библиотеку, как описано в разд. "Обработка библиотек, используемых драйвером" далее в этой главе.

### Чтобы очистить папку sources драйвера:

1. В окне среды сборки перейдите в папку sources драйвера.
2. В командном окне введите следующую команду:

```
staticdsv /clean
```

3. Чтобы очистить все библиотеки, введите следующую команду:

```
staticdsv /cleanalllibs
```

Команда `clean` не очищает файлы промежуточного представления, которые создаются для библиотеки при исполнении команды `staticdsv /lib`. Также команда `clean` не удаляет файл `Sdv-map.h`, если в нем флаг `Approved` имеет значение `True`. Эти файлы сохраняются для дальнейших сеансов верификации.

Более полную информацию по этому вопросу см. в разделе **Sdv-map.h** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80083>.

## Обработка библиотек, используемых драйвером

Включение библиотек в сеанс верификации SDV является весьма важным для определения, соблюдает ли драйвер SDV-правила. Например, если не включить код библиотеки, может казаться, что в драйвере пропущен требуемый вызов, который находится в библиотеке. Или драйвер может дублировать функцию, содержащуюся в библиотеке, что вызовет ошибку повторяющегося действия, например, двойное освобождение блокировки.

Чтобы включить библиотеку для верификации с драйвером, SDV сначала должен обработать эту библиотеку, чтобы подготовить внутреннее представление библиотеки для применения при верификации драйвера. Внутренне представление библиотеки сохраняется в файле с расширением LI, например, MyDrive.li.

Следует подвергать обработке все библиотеки, создаваемые для применения с разрабатываемым драйвером. Но предоставляющие функции интерфейса DDI библиотеки Windows (Wdm.lib, Wdf.lib, Bufferoverflow.lib, Hal.lib, Ntoskernel.lib, Wmi.lib и т. п.) обрабатывать не нужно, т. к. для верификации модель операционной системы предоставляет заглушки для функций DDI.

#### **Чтобы обработать библиотеку, применяемую в драйвере:**

1. В окне среды сборки выберите среду сборки для версии Windows, под которую разработан драйвер.

2. Перейдите в папку sources библиотеки.

3. В командном окне введите следующую команду:

```
staticdsv /clean
```

4. После чего введите следующую команду:

```
staticdsv /lib
```

Если SDV обнаруживает другие зависимости библиотеки, но не может найти код библиотеки, то он выводит предупреждающее сообщение: "Process <имя библиотеки>".

После того как SDV обработает библиотеку, он сохраняет файлы промежуточного представления для этой библиотеки и автоматически включает код библиотеки в сеансы верификации для всех драйверов, требующих данную библиотеку. Выполнять повторную обработку кода не требуется, если только код библиотеки не меняется.

Подробную информацию по этому предмету см. в разделе **Library Processing in Static Driver Verifier** (Обработка библиотек инструментом Static Driver Verifier) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80077>.

#### **Как отсканировать исходный код для создания файла Sdv-map.h**

Перед первым сеансом верификации драйвера необходимо отсканировать исходный код драйвера с помощью SDV. Во время сканирования SDV анализирует исходный код драйвера в поисках точек входа. Для драйверов KMDF точками входа являются объявления типов ролей функций обратного вызова драйвера, которые были описаны в разд. "Как аннотировать исходный код драйверов KMDF для SDV" ранее в этой главе.

#### **Файл Sdv-map.h**

SDV использует эти объявления для создания файла Sdv-map.h в каталоге исходного кода драйвера. В файле Sdv-map.h определены функции обратного вызова драйвера, значимые для SDV. Фактически Sdv-map.h соотносит релевантные функции обратного вызова с общими именами обратных вызовов, используемых в модели операционной системы SDV. Например, функция MyDpc обратного вызова MyDpc может быть соотнесена с именем fun\_WDF\_DPC\_1. Таким образом, SDV объединяет исходные коды драйвера с моделью операционной системы SDV в монолитную программу на языке C.

Рекомендуется выполнять необязательное сканирование исходного кода драйвера как отдельный шаг, особенно при первом сеансе верификации драйвера в SDV. Также желательно проверить файл Sdv-map.h, чтобы убедиться в том, что в нем определены все функции обратного вызова драйвера, которые требуется верифицировать. Если какой-либо из них нет в файле, ее можно добавить вручную.

### **Файл Sdv-map.h и число функций обратного вызова для каждого типа роли**

Для большинства функций обратного вызова SDV предполагает, что драйвер имеет самое большое одну функцию определенного типа роли. Но в действительности драйвер может иметь несколько функций обратного вызова для некоторых типов ролей, таких как, например, тип роли EVT\_WDF\_DPC. В данном случае SDV добавляет цифру к типу функции в файле Sdv-map.h.

Например, если драйвер имеет две функции обратного вызова DPC, то SDV соотносит их с именами fun\_WDF\_DPC\_1 и fun\_WDF\_DPC\_2.

SDV устанавливает максимальное число функций обратного вызова для типов ролей, которые могут иметь несколько функций обратного вызова. Если количество функций обратного вызова в драйвере превышает максимальное число, то это не делает драйвер недействительным, но усложняет применение SDV с данным драйвером.

Если число функций обратного вызова в драйвере превышает максимальное число, то SDV выдает сообщение о том, что в файле Sdv-map.h присутствуют повторяющиеся точки входа. Чтобы решить эту проблему, необходимо вручную отредактировать файл Sdv-map.h, определив в нем функции обратного вызова драйвера для каждого типа роли, с которым приходится работать SDV. Например, если в драйвере имеется восемь функций обратного вызова типа роли EVT\_WDF\_DPC (при максимально разрешенных в SDV семи), то необходимо сделать следующее:

1. Отредактировать файл Sdv-map.h, убрав определения fun\_WDF\_DPC\_5 по fun\_WDF\_DPC\_8.
2. Выполнить проверку драйвера в SDV.
3. Опять отредактировать файл Sdv-map.h, на этот раз определив типы fun\_WDF\_DPC\_5 по fun\_WDF\_DPC\_8 и убрав определения типов fun\_WDF\_DPC\_1 по fun\_WDF\_DPC\_4.
4. Опять выполнить проверку драйвера в SDV.

Список типов ролей, которые могут иметь несколько функций обратного вызова, приводится в разд. "Типы ролей функций обратного вызова KMDF для SDV" в конце данной главы, а в табл. 24.4 указывается максимальное число функций обратного вызова, поддерживаемое SDV для определенных типов ролей.

### **Шаги для сканирования исходного кода для создания файла Sdv-map.h**

Чтобы SDV мог использовать файл Sdv-map.h в сеансе верификации, первую строчку файла необходимо отредактировать следующим образом:

```
//Approved=true
```

### **Чтобы отсканировать исходный код для создания файла Sdv-map.h:**

1. В окне среды сборки перейдите в папку sources драйвера.
2. После чего введите следующую команду:

```
staticdsv /scan
```

Команда staticdsv /scan создает файл Sdv-map.h.

3. Просмотрите содержимое файла Sdv-map.h в папке драйвера sources, чтобы проверить точки входа (т. е. функции обратного вызова), которые SDV нашел в драйвере. Если требуется, отредактируйте определения точек входа.
4. Удостоверившись в правильности содержимого файла Sdv-map.h, отредактируйте первую строчку файла следующим образом:

```
//Approved=true
```

Для примера, в листинге 24.3 приведено содержимое файла Sdv-map.h для образца KMDF-драйвера Fail\_driver6.

#### **Листинг 24.3. Файл Sdv-map.h для образца драйвера Fail\_driver6**

```
//Approved=false
#define fun_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd
#define fun_WDF_DEVICE_CONTEXT_CLEANUP DeviceContextCleanUp
#define fun_WDF_IO_QUEUE_IO_READ EvtIoRead
#define fun_WDF_FILE_CONTEXT_CLEANUP_CALLBACK FileContextCleanup
#define fun_WDF_FILE_CONTEXT_DESTROY_CALLBACK FileContextDestroy
#define fun_WDF_DEVICE_CONTEXT_DESTROY DeviceContextDestroy
#define fun_WDF_IO_QUEUE_CONTEXT_CLEANUP_CALLBACK QueueCleanup
#define fun_WDF_IO_QUEUE_CONTEXT_DESTROY_CALLBACK QueueDestroy
#define fun_WDF_IO_QUEUE_IO_WRITE EvtIoWrite
#define fun_WDF_IO_QUEUE_IO_DEVICE_CONTROL EvtIoDeviceControl
```

Для получения дополнительной информации по этому вопросу см. раздел **Scanning the DriverEntry Routine** (Сканирование процедуры DriverEntry) на Web-сайте WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80057>.

## **Выполнение верификации**

После того как подготовлены файлы, можно выполнять верификацию всех правил, одного правила или набора определенных правил. Можно также выполнять верификацию по предопределенному списку правил.

Для получения подробностей о правилах SDV, которые можно указать для верификации драйверов KMDF, см. разд. "Правила KMDF для SDV" далее в этой главе.

### **Внимание!**

Никогда не запускайте несколько экземпляров SDV одновременно. SDV автоматически выполняет одновременную верификацию нескольких правил, используя число потоков, указанных в файле настроек.

#### **Чтобы выполнить верификацию всех правил:**

1. В окне среды сборки перейдите в папку sources драйвера.
2. После чего введите следующую команду:

```
staticcdv/rule:*
```

#### **Чтобы выполнить верификацию одного правила:**

1. В окне среды сборки перейдите в папку sources драйвера.

2. Введите команду staticdsv следующим образом:

```
staticdsv /rule:ИмяПравила
```

Здесь *ИмяПравила* указывает имя определенного правила SDV. Например, следующая команда выполняет верификацию правила OutputBufferAPI:

```
staticdsv /rule:outputbufferapi
```

#### Чтобы выполнить верификацию набора правил:

1. В окне среды сборки перейдите в папку sources драйвера.
2. Введите команду staticdsv следующим образом:

```
staticdsv /rule:Правило*
```

Здесь *Правило\** указывает набор правил на основе шаблона имени правила. Например, следующая команда выполняет верификацию набора правила семейства Request Buffer:

```
staticdsv /rule:bufafterreq*
```

В данном случае SDV выполняет верификацию следующих правил:

- ◆ BufAfterReqCompletedIntIoctl;
- ◆ BufAfterReqCompletedIntIoctlA;
- ◆ BufAfterReqCompletedIoctl;
- ◆ BufAfterReqCompletedIoctlA;
- ◆ BufAfterReqCompletedRead;
- ◆ BufAfterReqCompletedReadA;
- ◆ BufAfterReqCompletedWrite;
- ◆ BufAfterReqCompletedWriteA.

Список правил представляет собой обычный текстовый файл с расширением SDV, содержащий список правил. WDK предоставляет шаблоны SDV-файлов, содержащие часто используемые списки правил. Эти файлы находятся в папке %wdk%\tools\sdv\samples\rule\_sets. Содержимое одного из этих файлов (PnP.sdv) показано в листинге 24.4.

#### Листинг 24.4. Содержимое файла списка правил PnP.sdv

```
AddDevice  
PnpSameDeviceObject  
PnpIrpCompletion  
PnpSurpriseRemove  
TargetRelationNeedsRef
```

#### Чтобы выполнить верификацию списка правил:

1. В окне среды сборки перейдите в папку sources драйвера.
2. Введите команду staticdsv следующим образом:

```
staticdsv /config:СписокПравил.sdv
```

Здесь *СписокПравил.sdv* — имя SDV-файла, содержащего список правил.

Для получения исчерпывающей справочной информации по параметрам staticdsv см. раздел **Static Driver Verifier Commands** в документации набора разработчика WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80085>.

## Экспериментирование с SDV

Набор разработчика драйверов WDK содержит семь драйверов KMDF (Fail\_Driver1, ..., Fail\_Driver7) в папке %wdk%\tools\sdv\samples\fail\_drivers\kmdf. Эти драйверы разработаны специально, чтобы вызывать неудачное выполнение верификации SDV, с тем, чтобы можно было увидеть поведение SDV, когда он обнаруживает нарушения правил. Поэкспериментируйте на этих драйверах с верификацией правил SDV и KMDF.

### Внимание!

Так как образцы драйверов в папке \fail\_drivers\kmdf содержат преднамеренные ошибки, не используйте их в качестве основы для рабочих драйверов.

В листинге 24.5 приводится пример вывода команды staticdvc /rule:bufafterreq\* для обрата драйвера Fail\_driver6.

#### Листинг 24.5. Вывод результатов верификации набора правил KMDF Request Buffer для драйвера Fail\_driver6

```
C:\WINDDK\6001\tools\sdv\samples\fail_drivers\kmdf\
          fail_driver6\driver>staticdvc
/rule:"bufafterreq*"

Microsoft (R) Windows (R) Static Driver Verifier Version 1.5.314.0
Copyright (C) Microsoft Corporation. All rights reserved.

-----
Build   'driver'                                ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Scan    'driver'                                ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Compile 'driver'      for [sdv_harness_pnp_io_requests] ...Done
Link    'driver'      for [fail_library6.lib]     ...Done
Check   'driver'      for 'bufafterreqcompletedwrite' ...Running
Check   'driver'      for 'bufafterreqcompletedwritea' ...Running
Check   'driver'      for 'bufafterreqcompletedwritea' ...Done
Check   'driver'      for 'bufafterreqcompletedreada' ...Running
Check   'driver'      for 'bufafterreqcompletedreada' ...Done
Check   'driver'      for 'bufafterreqcompletedread' ...Running
Check   'driver'      for 'bufafterreqcompletedwrite' ...Done
Check   'driver'      for 'bufafterreqcompletedioctla' ...Running
Check   'driver'      for 'bufafterreqcompletedioctla' ...Done
```

```

Check  'driver'      for  'bufaftreqcompletedioctl'      ...Running
Check  'driver'      for  'bufaftreqcompletedread'       ...Done
Check  'driver'      for  'bufaftreqcompletedintioctla' ...Running
Check  'driver'      for  'bufaftreqcompletedioctl'      ...Done
Check  'driver'      for  'bufaftreqcompletedintioctV'   ...Running
Check  'driver'      for  'bufaftreqcompletedintioctla' ...Done
Check  'driver'      for  'bufaftreqcompletedintioctV'   ...Done
Static Driver Verifier performed 8 check(s) with:
2    Defect(s)
2    Rule Passes
4 Not Applicable Start Time :1/25/2007 2:35:56 PM and
End Time :1/25/2007 2:37:31 PM

```

## Просмотр отчетов SDV

После того как SDV завершит верификацию, он создает отчет об обнаруженных им дефектах. Для каждого дефекта SDV создает окно **Defect Viewer**, представляющее собой набор панелей, в которых выводится трассировка пути к дефекту. Список дефектов можно просмотреть с помощью окна **Static Driver Verifier Report Page**, в котором можно открыть окно **Defect Viewer** для просмотра результатов верификации.

В некоторых случаях трассировка ведет к дефектному оператору C, который можно рассматривать как основную причину ошибки. Но в более сложных случаях основную причину ошибки нельзя ассоциировать с одним оператором языка C. Рассмотрим, например, воображенное правило, приведенное в разд. *"А что внутри? Принцип работы движка верификации SDV"* ранее в этой главе:

- ◆ если путь исполнения кода в драйвере нарушает это правило, то SDV, возможно, сможет найти этот путь, но не сможет определить, было ли причиной нарушения вызов функции MyFunctionA или функции MyFunctionB;
- ◆ этот дефект будет ассоциирован с парой операторов: вызовом функции MyFunctionB и вызовом функции MyFunctionA.

### Чтобы открыть отчет SDV:

1. В окне среды сборки введите следующую команду:

```
staticdsv /view
```

2. В окне **Static Driver Verifier Report Page** выполните двойной щелчок по имени правила в узле **Defect(s)**, чтобы просмотреть информацию об ошибке в **Defect Viewer**.

SDV выводит правила, для которых были обнаружены нарушения, в узле **Defect(s)** в панели **Results**. На рис. 24.1 показан пример отчета SDV о результатах выполнения верификации правил семейства BufAfterReq\* для образца драйвера KMDF Fail\_driver6.

Результаты выполнения верификации SDV показываются в виде многоуровневого списка узлов. Узлы в списке представлены пиктограммами и ассоциированными текстовыми сообщениями. Узлы высшего уровня представляют категории результатов; наиболее важными из этих узлов являются узлы **Passed** (Годные), **Defect** (Дефектные), **Timeout** (Тайм-аут), **Spaceout** (Превышение лимита памяти), **Not Applicable** (Неприменимо) и **Uncertain** (Неопределенный). Узлы низкого уровня представляют результаты для каждого правила. В табл. 24.1 перечислены все существующие пиктограммы узлов и их значения.

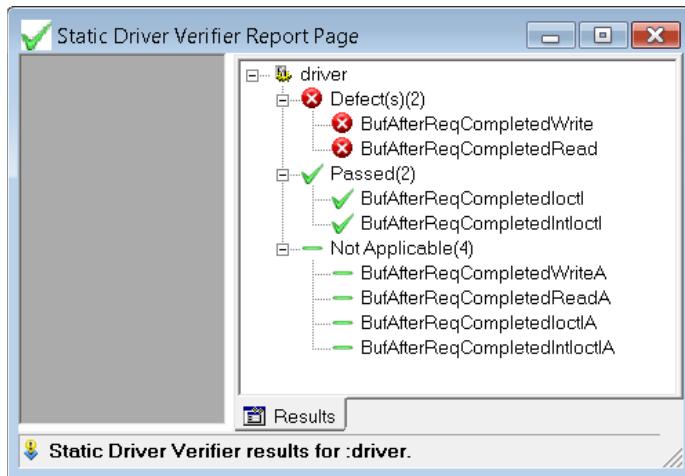


Рис. 24.1. Вывод отчета SDV в панели Results

Таблица 24.1. Пиктограммы, применяемые в выводе результатов верификации

| Значок        | Описание                                                                                                    |
|---------------|-------------------------------------------------------------------------------------------------------------|
| ✓             | Успех. SDV не смог доказать нарушение правила                                                               |
| ✗             | Дефект. SDV обнаружил одно нарушение правила                                                                |
| ✗             | Несколько дефектов. SDV обнаружил несколько нарушений                                                       |
| ⌚             | Тайм-аут. SDV прекратил выполнение верификации, т. к. был превышен временной лимит для правила              |
| MemoryWarning | Превышение лимита памяти. SDV прекратил выполнение верификации, т. к. был превышен лимит памяти для правила |
| —             | Неприменимо. Данное правило неприменимо к драйверу                                                          |
| ⚠             | Не одобрен файл распределения. Значение флага Approved в файле Sdv-map.h не установлено в True              |
| ⚠             | Инструментальная ошибка. Произошла внутренняя ошибка                                                        |
| ✓             | Неопределенность. SDV не смог выявить ни нарушения, ни соответствия правилу                                 |
| ✋             | Неподдерживаемая возможность. SDV не может интерпретировать один или несколько элементов кода драйвера      |

Для получения подробной информации о пиктограммах панели Results и о том, как использовать эту панель, см. раздел **Results Pane** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80081>.

## Утилита Defect Viewer

На рис. 24.2 показано окно утилиты Defect Viewer, в котором выведена информация о нарушении правила FDOPowerPolicyOwnerAPI в образце драйвера KMDF Fail\_Driver3. Описание шагов для сборки и верификации этого драйвера с помощью SDV приводится в разд. "Пример: пошаговый анализ драйвера Fail\_Driver3 с помощью SDV" далее в этой главе.

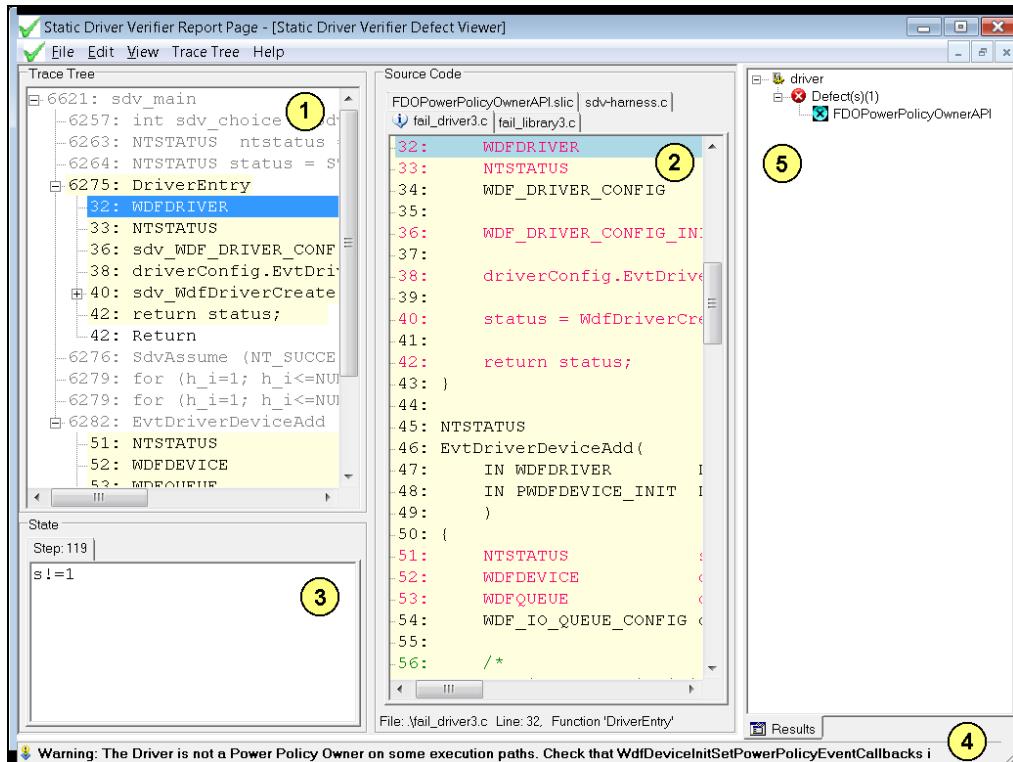


Рис. 24.2. Вывод информации о нарушении правила в Defect Viewer

На рис. 24.2 номерами обозначены элементы, предоставляющие информацию о результатах:

1. В панели **Trace Tree** выводится трассировка критических элементов исходного кода, которые были исполнены в пути нарушения правила.
2. В панели **Source Code** высвечиваются строки кода, соответствующие исходному коду в панели **Trace Tree**, в которой фрагменты трассировки выводятся красным шрифтом.

Каждая вкладка панели **Source Code** представляет шаг трассировки всего исходного кода в процессе верификации. Номер вкладки представляет последовательность данного шага в трассировке.

3. На вкладках в панели **State** выводятся булевые выражения для значений переменных драйвера, модели операционной системы SDV и правила.

SDV использует эти выражения для создания булевой абстракции, используемой в верификации. Если элемент исходного кода, выбранный в панели **Trace Tree** или **Source Code**, изменяет значения переменных таким образом, который вызывает изменение зна-

чений некоторых булевых переменных, тогда эти значения автоматически выводятся в панели **State**.

4. В строке состояния выводится описание дефекта.

5. Описание панели **Results** было приведено в объяснениях для рис. 24.1.

Подробную информацию о том, как использовать утилиту Defect Viewer для фильтрации и управления результатами верификации SDV, см. в разделе **Defect Viewer** в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80066>.

## Оптимальная методика: проверяйте результаты SDV

### Полезная информация

В некоторых случаях SDV выдает сообщения об ошибках, о которых можно подумать, что они не могут произойти в действительности. Такие сообщения иногда называются ложноположительными результатами. Но ложноположительное сообщение обычно указывает, что код является неопределенным каким-либо образом и зависит от предположения, о котором разработчик знает, что оно является действительным, но которое SDV не может верифицировать. Нужно всегда исследовать ложноположительные сообщения, чтобы проверить, что данные предположения оправданы. Оптимальным действием в такой ситуации будет следующее:

- если сообщение является ложноположительным, добавьте в код комментарий, информирующий последующих разработчиков, что данный вопрос был рассмотрен;
- если сообщение указывает действительную ошибку, то немедленно исправьте ее. Если же причина ошибки неизвестна, то исследуйте и протестируйте работу драйвера согласно трассировке дефекта, выданной в отчете SDV.

Если SDV выдает сообщение об ошибке, которое, как вам кажется, является неправильным, просмотрите ограничения SDV и ищите известные причины, которые могут вызывать неправильное интерпретирование исходного кода драйвера в SDV. В частности, SDV:

- ◆ полагает, что статически объявленный тип указателя всегда правильный и верно отражает его фактический динамический тип;
- ◆ не распознает, что 32-битные целые числа ограничены 32 битами; в результате SDV не обнаруживает ошибок переполнения и обнуления;
- ◆ использует 31 бит для представления целых чисел. Это ограничение может вызвать как ложноотрицательные, так и ложноположительные результаты;
- ◆ игнорирует адресную арифметику с указателями. Например, SDV не обнаруживает ситуации, в которых указатель увеличивается или уменьшается. Это ограничение также может вызвать как ложноотрицательные, так и ложноположительные результаты;
- ◆ игнорирует объединения;
- ◆ игнорирует операции на уровне битов. Эта неточность также может вызвать как ложноположительные, так и ложноотрицательные результаты;
- ◆ игнорирует операции приведения типов. SDV не обнаруживает ни ошибок, исправленных с помощью приведения, ни ошибок, вызванных приведением. Например, SDV полагает, что целочисленная переменная, приведенная к типу `char`, все еще имеет целочисленное значение;
- ◆ не может верифицировать функции обратного вызова драйвера, объявленные `static`. Эта проблема решается удалением ключевого слова `static` из объявления;

- ◆ не интерпретирует структурированную обработку исключений и не анализирует выражение и код для обработки исключения.

Для операторов `_try/_except` SDV анализирует защищенный раздел, как будто бы никакое исключение не вызывается, но не анализирует выражение и код для обработки исключения, как показано в следующем примере:

```
// Оператор try/except
_try {
    // Защищенный раздел.
}
_except (выражение) {
    // Обработчик исключения.
}
```

Для операторов `_try/_finally` SDV анализирует защищенный раздел, а после него — обработчик завершения, как будто бы никакое исключение не вызывается, как показано в следующем примере:

```
// Оператор try/finally
_try {
    // Защищенный раздел.
} _finally {
    // Обработчик завершения.
}
```

Как для операторов `_try/_except`, так и для операторов `_try/_finally` SDV игнорирует оператор `leave`;

- ◆ не может интерпретировать функции обратного вызова драйвера, определенные в драйвере экспорта, когда драйвер экспорта имеет файл определения модуля (с расширением DEF), который скрывает рабочую функцию драйвера.

Чтобы избежать этой проблемы, следует добавить рабочую функцию драйвера в раздел `EXPORTS` DEF-файла.

Подробную информацию по этому предмету см. в разделе **Static Driver Verifier Limitations** (Ограничения инструмента Static Driver Verifier) в документации WDK по адресу <http://go.microsoft.com/fwlink/?LinkId=80086>.

## Правила KMDF для SDV

В последующих разделах приводится описание правил KMDF для SDV. Правила разбиты по категориям, перечисленным в табл. 24.2.

**Таблица 24.2. Категории правил**

| Категория правил                          | Анализируемые типы функций DDI                         |
|-------------------------------------------|--------------------------------------------------------|
| Правила последовательности DDI            | Зависимые функции DDI                                  |
| Правила для инициализации устройств       | Функции DDI для инициализации объекта устройства       |
| Правила для очистки устройства управления | Функции DDI для создания и удаления объектов устройств |
| Правила для завершения запросов           | DDI-функции для объектов очередей и устройств          |

Таблица 24.2 (окончание)

| Категория правил                                   | Анализируемые типы функций DDI                                               |
|----------------------------------------------------|------------------------------------------------------------------------------|
| Правила для отмены запросов                        | DDI-функции для объектов запросов                                            |
| Правила для буферов, списков MDL и памяти запросов | DDI-функции для доступа к памяти объектов запросов                           |
| Правила для владельцев политики энергопотребления  | Функции DDI для драйверов, являющихся владельцами политики энергопотребления |

### Примечание

Это всего лишь обзорные сведения, призванные дать введение в правила KMDF. Для получения исчерпывающей информации о правилах KMDF для SDV см. документацию набора разработчика драйверов WDK.

## Правила последовательности функций DDI для KMDF

Эти правила указывают, что определенные функции DDI должны вызываться перед другими зависимыми функциями DDI. Например, правило FileObjectConfigured указывает, что файловый объект конфигурируется в драйвере, вызывая функцию `WdfDeviceInitSetFileObjectConfig` перед вызовом функции `WdfRequestGetFileObject` для того же самого файлового объекта.

### ◆ Правило DriverCreate.

Указывает, что драйвер KMDF вызывает функцию `WdfDriverCreate` из своей функции `DriverEntry`.

### ◆ Правило FileObjectConfigured.

Указывает, что перед функцией `WdfRequestGetFileObject` вызывается функция `WdfDeviceInitSetFileObjectConfig`.

### ◆ Правило CtlDeviceFinishInitDeviceAdd.

Указывает, что если драйвер Plug and Play создает объект устройства управления в функции обратного вызова `EvtDriverDeviceAdd`, то после создания этого объекта и перед выходом из функции `EvtDriverDeviceAdd` он должен вызывать функцию `WdfControlFinishInitialization`. Это правило не распространяется на драйверы, не поддерживающие Plug and Play.

### ◆ Правило CtlDeviceFinishInitDrEntry.

Указывает, что если драйвер Plug and Play создает объект устройства управления в функции `DriverEntry`, то после создания этого объекта и перед выходом из функции `DriverEntry` он должен вызывать функцию `WdfControlFinishInitialization`. Это правило не распространяется на драйверы, не поддерживающие Plug and Play.

## Правила инициализации устройств для KMDF

В общих чертах, эти правила проверяют, чтобы функции DDI для инициализации объекта устройства вызывались перед созданием данного объекта. Например, правило `DeviceInitAllocate` указывает, что для объекта PDO или для объекта устройства управления метод `WdfPdoInitAllocate` или `WdfControlDeviceInitAllocate` для инициализации инфраструкту-

турного объекта устройства необходимо вызывать перед вызовом метода `WdfDeviceCreate` для этих объектов.

◆ Правило `DeviceInitAllocate`.

Указывает, что для объекта PDO или для объекта устройства управления метод `WdfPdoDeviceInitAllocate` или `WdfControlDeviceInitAllocate` для инициализации инфраструктурного объекта устройства необходимо вызывать перед вызовом метода `WdfDeviceCreate` для этих объектов.

◆ Правило `DeviceInitAPI`.

Указывает, что инфраструктурные методы инициализации объекта FDO `WdfDeviceInitXxx` и `WdfFdoInitXxx` должны вызываться перед вызовом метода `WdfDeviceCreate` для объекта устройства.

◆ Правило `ControlDeviceInitAPI`.

Указывает, что для объекта устройства управления методы `WdfDeviceInitXxx` и `WdfControlDeviceInitSetShutdownNotification` для инициализации инфраструктурного объекта устройства необходимо вызывать перед вызовом драйвером метода `WdfDeviceCreate`.

◆ Правило `PdoDeviceInitAPI`.

Указывает, что для объекта PDO инфраструктурные методы инициализации объекта устройства `WdfDeviceInitXxx` и инфраструктурные методы инициализации устройства `WdfPdoInitXxx` должны вызываться перед вызовом драйвером метода `WdfDeviceCreate`.

## Правила очистки устройства управления для KMDF

Эти правила указывают, что созданный драйвером объект устройства управления должен быть в необходимое время удален должным образом. Например, правило `ControlDeviceDeleted` указывает, что любой драйвер Plug and Play, создающий объект устройства управления, должен в одной из своих функций обратного вызова очистки для устройства удалить этот объект перед выгрузкой драйвера.

◆ Правило `Cleanup4CtlDeviceRegistered`.

Указывает, что если драйвер Plug and Play создает объект устройства управления, то он должен зарегистрировать или функцию обратного вызова `EvtCleanupCallback` или `EvtDestroyCallback` в структуре `WDF_OBJECT_ATTRIBUTES` для этого объекта, или зарегистрировать функцию обратного вызова `EvtDeviceSelfManagedToCleanup` в структуре `WDF_PNPPOWER_EVENT_CALLBACKS`.

◆ Правило `ControlDeviceDeleted`.

Указывает, что если драйвер Plug and Play создает объект устройства управления, то драйвер должен удалить этот объект в одной из своих функций обратного вызова очистки перед выгрузкой драйвера.

## Правила завершения запроса для KMDF

Эти правила указывают, что каждый запрос, доставленный драйверу в одной из функций обратного вызова в очередь по умолчанию (`EvtTimerFunc`, `EvtDpcFunc`, `EvtInterruptDpc`, `EvtInterruptEnable`, `EvtInterruptDisable`, `EvtWorkItem`), нужно завершить только один раз.

Например, правило DeferredRequestCompleted обуславливает, что если запрос не завершен в функции обратного вызова очереди по умолчанию, а был отложен для дальнейшей обработки, этот запрос должен быть завершен в функции обратного вызова отложенной обработки, за исключением случаев, когда запрос был переслан и доставлен инфраструктуре или когда вызывается метод `WdfRequestStopAcknowledge`.

◆ Правило DoubleCompletion.

Обуславливает, что драйверы не завершают запрос дважды.

◆ Правило DoubleCompletionLocal.

Обуславливает, что драйверы не завершают запрос дважды. Это такое же правило, как и правило DoubleCompletion, за исключением того, что с целью оптимизации проверка выполняется только внутри обработчиков запроса очереди ввода/вывода по умолчанию.

◆ Правило RequestCompleted.

Указывает, что все запросы, доставленные в очередь ввода/вывода по умолчанию драйвера, должны быть завершены, за исключением случаев, когда драйвер откладывает или пересыпает запрос или когда вызывает для запроса метод `WdfRequestStopAcknowledge`. Это правило дает результат "Неприменимо" для драйверов, не регистрирующихся для любого из следующих:

- любых релевантных функций обратного вызова очистки или удаления для объекта устройства, очереди или файла;
- функций обратного вызова Plug and Play и энергопотребления, таких как `EvtDeviceSelfManagedIoCleanup` или `EvtDeviceShutdownNotification`;
- функции обратного вызова `EvtDriverUnload`.

◆ Правило RequestCompletedLocal.

Предупреждает о возможном дефекте, если запрос не завершен в любой из функций обратного вызова очереди ввода/вывода по умолчанию и если запрос не помечен в этих функциях как отменяемый. Это правило предназначено только для драйверов, для которых неприменимо правило RequestCompleted.

◆ Правило DeferredRequestCompleted.

Обуславливает, что если запрос, представленный функции обратного вызова очереди ввода/вывода по умолчанию драйвера, не завершен в этой функции, а был отложен для дальнейшей обработки, этот запрос должен быть завершен в функции обратного вызова отложенной обработки, за исключением случаев, когда запрос был переслан и доставлен инфраструктуре или когда вызывается метод `WdfRequestStopAcknowledge`.

## Правила отмены запроса для KMDF

Эти правила определяют правильный стиль программирования для отмены запроса, который был доставлен драйверу в одной из функций обратного вызова стандартной очереди ввода/вывода и помеченный как отменяемый методом `WdfRequestMarkCancelable`. Например, правило `ReqNotCanceled` обуславливает, что если драйвер пометил запрос как отменяемый, а потом завершает его в функции обратного вызова отложенной обработки, то перед завершением запроса драйвер должен вызвать для него метод `WdfRequestUnmarkCancelable`, и запрос может быть завершен только в том случае, если он еще не был отменен.

◆ Правило ReqNotCanceled.

Указывает, что если запрос, помеченный как отменяемый, завершается в функции обратного вызова отложенной обработки, то перед завершением запроса для него необходимо вызвать метод `WdfRequestUnmarkCancelable`. Запрос можно завершить только в том случае, если он еще не был отменен.

◆ Правило ReqNotCanceledLocal.

Указывает, что если запрос, помеченный как отменяемый, завершается в функции обратного вызова очереди ввода/вывода по умолчанию, то перед завершением запроса для него необходимо вызвать метод `WdfRequestUnmarkCancelable`, и запрос может быть завершен только в том случае, если он еще не был отменен. Это правило отличается от правила `ReqNotCanceled` тем, что в нем указывается необходимость выполнять проверку из обработчиков запросов для очереди ввода/вывода по умолчанию.

◆ Правило ReqIsNotCancelable.

Указывает, что только отменяемые отложенные запросы можно делать неотменяемыми перед их завершением. Иными словами, если драйвер вызывает для запроса метод `WdfRequestUnmarkCancelable`, то перед этим он должен вызывать метод `WdfRequestUnmarkCancelable`.

◆ Правило MarkCancOnCancReq.

Указывает, что метод `WdfRequestMarkCancelable` нельзя вызывать два раза подряд для одного и того же запроса.

◆ Правило MarkCancOnCancReqLocal.

Указывает, что метод `WdfRequestMarkCancelable` нельзя вызывать два раза подряд для одного и того же запроса. Данное правило определяет проверку только в функциях обратного вызова очереди ввода/вывода по умолчанию.

◆ Правило ReqIsCancOnCancReq.

Указывает, что метод `WdfRequestIsCanceled` можно вызывать только для запросов, не помеченных как отменяемые.

## Правила для буферов, списков MDL и памяти запросов

Эти правила указывают, что после завершения запроса нельзя обращаться к его буферу, списку MDL или объекту памяти. Например, правило `BufAfterReqCompletedRead` задает эту проверку для случаев, когда к буферу выполняется обращение из функции обратного вызова `EvtIoRead`.

### Примечание

Для данного множества KMDF-правил SDV рассматривает 14 функций DDI как возможные функции доступа к буферу, 15 функций — как возможные функции доступа к списку MDL, и 10 функций — как возможные функции доступа к памяти. Кроме функций, перечисленных в данном примере, правило `BufAfterReqCompletedReadA` определяет проверку для 10 других функций DDI, которые обращаются к буферу.

◆ Правило InputBufferAPI.

Указывает, чтобы в функции обратного вызова `EvtIoRead` использовались правильные функции DDI для обращения к буферам, спискам MDL и объектам памяти. В частности, в данном случае нельзя вызывать следующие функции DDI:

`WdfRequestRetrieveInputBuffer`, `WdfRequestRetrieveUnsafeUserInputBuffer`,  
`WdfRequestRetrieveInputWdmMdl` и `WdfRequestRetrieveInputMemory`.

◆ Правило `OutputBufferAPI`.

Указывает, чтобы в функции обратного вызова `EvtIoWrite` использовались правильные функции DDI для обращения к буферам и объектам памяти. В частности, в данном случае нельзя вызывать следующие функции DDI:

`WdfRequestRetrieveOutputBuffer`, `WdfRequestRetrieveUnsafeUserOutputBuffer`,  
`WdfRequestRetrieveOutputWdmMdl` и `WdfRequestRetrieveOutputMemory`.

## Правила для буферов запросов

◆ `BufAfterReqCompletedRead` и `BufAfterReqCompletedReadA`.

Указывают, что в функции `EvtIoRead` нельзя обращаться к буферу после завершения запроса. Доступ к буферу выполняется с помощью методов `WdfRequestRetrieveOutputBuffer` или `WdfRequestRetrieveUnsafeUserOutputBuffer`.

◆ Правила `BufAfterReqCompletedWrite` и `BufAfterReqCompletedWriteA`.

Указывают, что в функции `EvtIoWrite` нельзя обращаться к буферу после завершения запроса. Доступ к буферу выполняется с помощью методов `WdfRequestRetrieveInputBuffer` или `WdfRequestRetrieveUnsafeUserInputBuffer`.

◆ Правила `BufAfterReqCompletedIoctl` и `BufAfterReqCompletedIoctlA`.

Указывают, что в функции `EvtIoDeviceControl` нельзя обращаться к буферу после завершения запроса. Доступ к буферу выполняется с помощью методов `WdfRequestRetrieveOutputBuffer`, `WdfRequestRetrieveUnsafeUserOutputBuffer`,  
`WdfRequestRetrieveInputBuffer` или `WdfRequestRetrieveUnsafeUserInputBuffer`.

◆ Правила `BufAfterReqCompletedIntIoctl` и `BufAfterReqCompletedIntIoctlA`.

Указывают, что в функции `EvtIoInternalDeviceControl` нельзя обращаться к буферу после завершения запроса. Доступ к буферу выполняется с помощью методов `WdfRequestRetrieveOutputBuffer`, `WdfRequestRetrieveUnsafeUserOutputBuffer`,  
`WdfRequestRetrieveInputBuffer` или `WdfRequestRetrieveUnsafeUserInputBuffer`.

## Правила для запросов обращения к спискам MDL

◆ Правила `MdlAfterReqCompletedRead` и `MdlAfterReqCompletedReadA`.

Указывают, что в функции `EvtIoRead` нельзя обращаться к объекту списка MDL после завершения запроса. Доступ к списку MDL выполняется с помощью метода `WdfRequestRetrieveOutputWdmMdl`.

◆ Правила `MdlAfterReqCompletedWrite` и `MdlAfterReqCompletedWriteA`.

Указывают, что в функции `EvtIoWrite` нельзя обращаться к объекту списка MDL после завершения запроса. Доступ к списку MDL выполняется с помощью метода `WdfRequestRetrieveInputWdmMdl`.

◆ Правила `MdlAfterReqCompletedIoctl` и `MdlAfterReqCompletedIoctlA`.

Указывают, что в функции `EvtIoDeviceControl` нельзя обращаться к объекту списка MDL после завершения запроса. Доступ к списку MDL выполняется с помощью методов `WdfRequestRetrieveInputWdmMdl` или `WdfRequestRetrieveOutputWdmMdl`.

◆ Правила MdlAfterReqCompletedIntIoctl и MdlAfterReqCompletedIntIoctlA.

Указывают, что в функции *EvtIoInternalDeviceControl* нельзя обращаться к объекту списка MDL после завершения запроса. Доступ к списку MDL выполняется с помощью методов *WdfRequestRetrieveInputWdmMdl* или *WdfRequestRetrieveOutputWdmMdl*.

### Правила для запросов обращения к памяти

◆ Правила MemAfterReqCompletedRead и MemAfterReqCompletedReadA.

Указывают, что в функции *EvtIoRead* нельзя обращаться к объекту памяти после завершения запроса. Доступ к объекту памяти выполняется с помощью метода *WdfRequestRetrieveInputWdmMdl*.

◆ Правила MemAfterReqCompletedWrite и MemAfterReqCompletedWriteA.

Указывают, что в функции *EvtIoWrite* нельзя обращаться к объекту памяти после завершения запроса. Доступ к объекту памяти выполняется с помощью метода *WdfRequestRetrieveInputMemory*.

◆ Правила MemAfterReqCompletedIoctl и MemAfterReqCompletedIoctlA.

Указывают, что в функции *EvtIoDeviceControl* нельзя обращаться к объекту памяти после завершения запроса. Доступ к объекту памяти выполняется с помощью метода *WdfRequestRetrieveInputMemory* или *WdfRequestRetrieveOutputMemory*.

◆ Правила MemAfterReqCompletedIntIoctl и MemAfterReqCompletedIntIoctlA.

Указывают, что в функции *EvtIoInternalDeviceControl* нельзя обращаться к объекту памяти после завершения запроса. Доступ к объекту памяти выполняется с помощью методов *WdfRequestRetrieveInputMemory* или *WdfRequestRetrieveOutputMemory*.

### Правила DDI для владельцев политики энергопотребления

Эти правила указывают, что драйвер, не являющийся владельцем политики энергопотребления, не может вызывать следующие три функции DDI управления энергопотреблением:

*WdfDeviceInitSetPowerPolicyEventCallbacks*, *WdfDeviceAssignSOIdleSettings* и  
*WdfDeviceAssignSxWakeSettings*.

Например, правило *NonPnPDrvPowerPolicyOwner* указывает, что драйвер, не поддерживающий Plug and Play, не может вызывать эти три функции DDI.

Данное множество правил включает два правила предварительного условия: *FDODriver* и *NotPowerPolicyOwner*. В табл. 24.3 приводятся правила, которые зависят от правил предварительного условия, и результаты правил предварительного условия, которые определяют, применимо ли зависимое правило к драйверу. Если правило зависит больше чем от одного предварительного условия, то для того, чтобы правило было применимо к драйверу, должны присутствовать оба результата.

**Таблица 24.3. Правила предварительного условия для правил PowerPolicyOwner**

| Правила PowerPolicyOwner            | Правила предварительного условия |                            |
|-------------------------------------|----------------------------------|----------------------------|
|                                     | <i>FDODriver</i>                 | <i>NotPowerPolicyOwner</i> |
| <i>NonFDONotPowerPolicyOwnerAPI</i> | Fail (Неудача)                   | Pass (Успех)               |
| <i>FDOPowerPolicyOwnerAPI</i>       | Pass (Успех)                     | Неприменимо                |

◆ Правило FDODriver (предварительное условие).

Указывает, является ли драйвер функциональным драйвером: если перед возвращением из своей функции обратного вызова *EvtDriverDeviceAdd* драйвер вызывает метод *WdfFdoInitSetFilter*, то драйвер является не функциональным драйвером, а драйвером фильтра. Если правило представляет результат Pass, то драйвер является функциональным драйвером; если же результат Fail или Неприменимо, то драйвер не является функциональным драйвером. Например, это правило представляет результат Неприменимо для любого драйвера, не поддерживающего Plug and Play, т. е. любого драйвера, который не предоставляет функцию обратного вызова *EvtDriverDeviceAdd*.

Это правило предварительного условия для правил NonFDONotPowerPolicyOwnerAPI и FDOPowerPolicyOwnerAPI. Любой дефект, выявленный правилом, в действительности указывает, что к драйверу применимо правило NonFDONotPowerPolicyOwnerAPI; результат Pass, выявленный правилом, в действительности указывает, что к драйверу применимо правило FDOPowerPolicyOwnerAPI.

◆ Правило NotPowerPolicyOwner (предварительное условие).

Указывает, какой драйвер является владельцем политики энергопотребления.

- Если вызывается функция обратного вызова *EvtDriverDeviceAdd*, но не вызывается метод *WdfFdoInitSetFilter*, то драйвер является функциональным драйвером и, следовательно, владельцем политики энергопотребления.
- Если драйвер вызывает метод *WdfPdoInitAssignRawDevice*, то драйвер является "сырым" драйвером PDO и, следовательно, владельцем политики энергопотребления.

Если драйвер не является владельцем политики энергопотребления, то результатом применения правила является Pass.

Если драйвер является владельцем политики энергопотребления, то результатом правила является Fail или же правило неприменимо.

Это правило предварительного условия для правила NonFDONotPowerPolicyOwnerAPI. Любой неудачный результат, выдаваемый этим правилом, в действительности указывает, что к драйверу применимо правило NonFDONotPowerPolicyOwnerAPI.

◆ Правило NonFDONotPowerPolicyOwnerAPI.

Указывает, что если не-FDO-драйвер не является владельцем политики энергопотребления, то тогда три функции DDI, связанные с управлением энергопотребления, нельзя вызывать. Чтобы это правило было применимо, необходимо чтобы были удовлетворены следующие два предварительные условия:

- правило FDODriver должно выдавать результат Fail;
- правило NotPowerPolicyOwner должно выдавать результат Pass.

◆ Правило NonPnPDrvPowerPolicyOwnerAPI.

Указывает, что драйвер, не поддерживающий Plug and Play, не может вызывать три функции DDI, связанные с управлением энергопотребления. Для PnP-драйверов и не-PnP-драйверов без дефектов, это правило дает результат Неприменимо.

◆ Правило FDOPowerPolicyOwnerAPI.

Предупреждает, если драйвер FDO, который вызывает три функции DDI, связанные с управлением энергопотребления, уступает владение политикой энергопотребления, т. к.

эти функции можно вызывать только в тех ветвях исполнения, для которых драйвер является владельцем политики энергопотребления. Чтобы это правило было применимо, необходимо чтобы было удовлетворено следующее предварительное условие: результатом применения правила FDO Driver должно быть Pass.

### **Иерархии и предварительные условия для правил – решение проблемы, представляющей сложными правилами**

Когда я начала участвовать в работе над проектом в 2004 г., KMDF все еще находилась в стадии разработки, а SDV уже использовался для проверки драйверов WDM. Я видела преимущества новой инфраструктуры, т. к. она вносила более высокий уровень абстракции, которая позволила бы разработчикам драйверов сэкономить время на написании кода и концентрироваться на специфичных для драйвера вопросах. Это не только уменьшает размер драйвера, но также облегчает процесс его отладки. В то же самое время, KMDF налагает довольно нетривиальные правила использования интерфейса DDI. Было ясно, что предоставление разработчикам драйверов какого-либо способа для автоматической проверки драйверов KMDF на удовлетворение правилам применения интерфейса DDI очень бы помогло им в переходе на новую инфраструктуру.

SDV очень здорово проверяет правила использования интерфейса DDI, как было продемонстрировано его применением для проверки драйверов WDM. Это было моей мотивацией для разработки правил SDV для KMDF. Разработка правил SDV для KMDF было нелегкой задачей, требующей дальнейшего усовершенствования методов написания правил.

Так как по сравнению с WDM уровень функций DDI для KMDF более высокий, то многие правила для KMDF оказались более сложными, чем большинство правил для WDM. Обычно в одном правиле для KMDF вовлечено больше функций DDI, и необходимо выражать более сложные зависимости между ними. Но с другой стороны, для целей производительности весьма желательно, чтобы каждое правило было максимально простым. Эта проблема была решена путем разбиения сложного правила на несколько более простых правил, что по существу является разбиением одной машины состояний на несколько более простых машин состояний. Кроме этого, некоторые правила применимы только к драйверам со специфическими свойствами. Эта проблема была решена введением иерархии правил и правил предварительного условия, что было новым понятием для SDV.

Этот проект предоставил мне уникальную возможность работать с двумя великолепными командами — командой разработчиков KMDF и командой разработчиков SDV — и участвовать в разработке новой современной технологии для разработки драйверов. (Элла Баунимова (*Ella Bouнимова*), член команды разработчиков *Static Analysis Tools for Drivers, Microsoft*.)

## **Пример: пошаговый анализ драйвера Fail\_Driver3 с помощью SDV**

В этом разделе приводится пошаговое рассмотрение применения SDV для анализа образца драйвера KMDF Fail\_Driver3. Образец драйвера Fail\_Driver3 содержит преднамеченные ошибки для демонстрации применения SDV и интерпретирования полученных результатов. Кроме файла исходного кода и заголовочного файла для самого драйвера, в образце драйвера Fail\_Driver3 используется библиотека, содержащая один файл исходного кода и один заголовочный файл.

Образец драйвера Fail\_Driver3 поставляется с набором разработчика WDK и находится в папке %wdk%\tools\sdv\samples\fail\_drivers\kmdf\fail\_driver3.

## Подготовка к верификации драйвера Fail\_Driver3

Процесс компоновки и верификации образца драйвера Fail\_Driver3 с помощью SDV состоит из шагов, описанных в следующих разделах.

### Сборка библиотеки драйвера Fail\_Driver3 с помощью SDV

Чтобы начать процесс верификации драйвера Fail\_Driver3, необходимо перейти в папку, содержащую библиотеку драйвера, и выполнить ее сборку. Дефекты в драйвере Fail\_Driver3 могут быть обнаружены только тогда, когда он скомпонован со своей библиотекой.

**Чтобы выполнить сборку библиотеки драйвера Fail\_Driver3 в SDV:**

1. Откройте окно среды сборки, выполнив последовательность команд меню: **Start | All Programs | Windows Driver Kits | Версия\_WDK | Build Environment | Операционная система | Среда сборки.**

SDV поддерживает все свободные и проверочные версии сред сборки. Для верификации этого образца драйвера необходимо выбрать свободную сборку.

2. В окне среды сборки перейдите в следующую папку:

`%wdk%\tools\sdv\samples\fail_drivers\kmdf\fail_driver3\library`

3. Для сборки проекта выполните следующие команды:

```
staticdv /clean  
staticdv /lib
```

При выполнении команды `staticdv /lib` SDV вызывает утилиту Build для компилирования и сборки библиотеки для внешнего использования и создает файлы, которые необходимо включить в библиотеку для выполнения верификации драйвера.

### Создание файла Sdv-map.h для драйвера Fail\_Driver3

Прежде чем SDV может выполнить верификацию драйвера, он должен получить информацию о поддерживаемых драйвером возможностях. При первом запуске SDV автоматически сканирует исходный код драйвера в поисках точек входа. Для драйверов KMDF точками входа являются объявления типов ролей функций обратного вызова драйвера, которые были описаны в разд. "Как аннотировать исходный код драйверов KMDF для SDV" ранее в этой главе. Этот шаг можно также выполнить вручную.

Драйвер Fail\_Driver3 необходимо отсканировать явным образом, чтобы обнаружить все функции обратного вызова. SDV сохраняет результаты сканирования драйвера в файл Sdv-map.h, который он создает в папке sources драйвера.

#### Внимание!

Проверьте созданный файл Sdv-map.h, чтобы убедиться в правильности обнаруженных функций обратного вызова. Проверенный файл можно одобрить, чтобы SDV сохранил его для будущих верификаций.

**Чтобы выполнить сканирования драйвера Fail\_driver3 и создать файл Sdv-map.h:**

1. В окне среды сборки перейдите в папку драйвера:

`%wdk%\tools\sdv\samples\fail_drivers\kmdf\fail_driver3\driver`

- Выполните сканирование функций обратного вызова драйвера Fail\_Driver3, введя следующую команду в окне среды сборки:

```
staticdsv /scan
```

- Откройте файл Sdv-map.h для просмотра, для чего введите следующую команду в окне среды сборки:

```
notepad sdv-map.h
```

Файл Sdv-map.h содержит следующие операторы #define для функций обратного вызова, которые были обнаружены при сканировании драйвера Fail\_Driver3:

```
//Approved=false
#define fun_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd
#define fun_WDF_IO_QUEUE_IO_READ EvtIoRead
#define fun_WDF_TIMER_1 EvtTimerFunc
#define fun_WDF_DRIVER_UNLOAD EvtDriverUnload
#define fun_WDF_REQUEST_CANCEL EvtRequestCancel
#define fun_WDF_IO_QUEUE_IO_WRITE EvtIoWrite
#define fun_WDF_IO_QUEUE_IO_DEVICE_CONTROL EvtIoDeviceControl
```

#### Чтобы одобрить файл Sdv-map.h:

- Откройте файл Sdv-map.h в текстовом редакторе Notepad (Блокнот) и установите значение флага //Approved B true.
- Сохраните и закройте файл.

## Выполнение верификации драйвера Fail\_Driver3

В образец драйвера Fail\_Driver3 была преднамеренно вставлена ошибка, которая вызывает нарушение правила FDOPowerPolicyOwnerAPI. Правило FDOPowerPolicyOwnerAPI применяется только к функциональным драйверам. Поэтому необходимо сначала проверить, что драйвер Fail\_Driver3 является функциональным драйвером, для чего выполняется проверка правила предварительного условия FDODriver.

Чтобы проверить, что драйвер Fail\_Driver3 является драйвером FDO, в окне среды сборки введите следующую команду:

```
staticdsv /rule:FDODriver
```

Результаты выполнения этой команды показаны в листинге 24.6. Как можно видеть из этого листинга, драйвер Fail\_Driver3 прошел проверку на соответствие этому правилу с результатом Pass. Это означает, что к нему применимо правило FDOPowerPolicyOwnerAPI.

#### Листинг 24.6. Результаты проверки драйвера Fail\_Driver3 на требование правила FDODriver

```
C:\WINDDK\6001\tools\sdv\samples\fail_drivers\kmdf\fail_driver3\driver>
staticdsv /rule:FDODriver
```

```
Microsoft (R) Windows (R) Static Driver Verifier Version 1.5.314.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Build      'driver'                      ...Done
Link      'driver'  for  [fail_library3.lib]  ...Done
```

```

Scan      'driver'                                ...Done
Compile   'driver'  for  [sdv_harness_pnp_io_requests]  ...Done
Link      'driver'  for  [fail_library3.lib]          ...Done
Check     'driver'  for  'fdodriver'                 ...Running
Check     'driver'  for  'fdodriver'                 ...Done
Static Driver Verifier performed 1 check(s) with:
  1 Rule Passes
Start Time  :1/25/2007 3:34:06 PM and End Time  :1/25/2007 3:34:16 PM

```

**Чтобы очистить папку для верификации другого правила, в окне среды сборки введите следующую команду:**

```
staticcdv /clean
```

На этот раз, проверьте, что драйвер Fail\_Driver3 соответствует требованиям правила FDOPowerPolicyOwnerAPI.

**Чтобы проверить, что драйвер Fail\_Driver3 правильно вызывает три функции DDI для управления энергопотреблением, в окне среды сборки введите следующую команду:**

```
staticcdv /rule:FDOPowerPolicyOwnerAPI
```

Результаты выполнения этой команды показаны в листинге 24.7. Как можно видеть из этого листинга, драйвер Fail\_Driver3 содержит дефект. Это означает, что драйвер Fail\_Driver3 прошел проверку на соответствие правилу FDOPowerPolicyOwnerAPI с результатом Fail, что в свою очередь означает, что драйвер не вызывает функции DDI для управления энергопотреблением должным образом.

#### Листинг 24.7. Результаты проверки драйвера Fail\_Driver3 на соответствие правилу FDOPowerPolicyOwnerAPI

```

C:\WINDDK\6001\tools\sdv\samples\fail_drivers\kmdf\fail_driver3\driver> staticcdv
/rule:"FDOPowerPolicyOwnerAPI"
-----
Microsoft (R) Windows (R) Static Driver Verifier Version 1.5.314.0
Copyright (C) Microsoft Corporation. All rights reserved.
-----
Build  'driver'    ...Done
Link   'driver'    for  [fail_library3.lib]          ...Done
Scan   'driver'                                ...Done
Compile 'driver'    for  [sdv_harness_pnp_io_requests]  ...Done
Link   'driver'    for  [fail_library3.lib]          ...Done
Check   'driver'   for  'fdopowerpolicyownerapi'    ...Running
Check   'driver'   for  'fdopowerpolicyownerapi'    ...Done
Static Driver Verifier performed 1 check(s) with:
  1  Defect(s)
Start Time  :1/25/2007 3:45:43 PM and End Time  :1/25/2007 3:45:54 PM

```

## Просмотр результатов верификации драйвера Fail\_Driver3

Результаты верификации образцов драйверов можно с удобством просмотреть с помощью утилиты Static Driver Defect Report.

### Чтобы просмотреть отчет о дефектах для драйвера Fail\_Driver3:

1. В окне среды сборки введите следующую команду:

```
staticcdv /view
```

2. В панели **Results** окна **Static Driver Verifier Report Page** выполните двойной щелчок по правилу FDOPowerPolicyOwnerAPI.

Это откроет окно утилиты Defect Viewer, в котором выводится трассировка ветви кода к точке нарушения правила. Окно утилиты Defect Viewer с результатами анализа для образца драйвера Fail\_Driver3 было показано ранее в этой главе на рис. 24.2.

### Ознакомление с нарушенными правилами

Прежде чем начинать поиск в исходном коде источника нарушения правила, будет хорошей идеей ознакомиться с описанием нарушенных правил, просмотрев их в соответствующей странице справки. После этого вы сможете более уверенно разобраться с кодом, вызвавшим нарушение.

Чтобы просмотреть код для нарушенного правила, в панели **Source Code** окна **Static Driver Verifier Defect Viewer** выберите вкладку для файла FDOPowerPolicyOwnerAPI.slic.

### Пошаговый просмотр трассировки нарушения правила в драйвере Fail\_Driver3

Драйвер Fail\_Driver3 нарушает правило FDOPowerPolicyOwnerAPI. С помощью панели **Trace Tree** путь исполнения можно просмотреть пошагово. При прокрутке элементов в панели **Trace Tree**:

- ◆ файл исходного кода, в котором находятся выбранные элементы, передвигается к верху списка файлов в панели **Source Code**;
- ◆ высвечивается связанный линия кода.

При просмотре нарушения правила FDOPowerPolicyOwnerAPI для образца драйвера Fail\_Driver3 следует обратить внимание на следующее.

- ◆ В своей функции обратного вызова `WDF_DRIVER_DEVICE_ADD` драйвер Fail\_Driver3 вызывает библиотечную функцию `SDVTest_wdf_FDOPowerPolicyOwnerAPI`.
- ◆ Внутри этой библиотечной функции функция `WdfDeviceInitSetPowerPolicyOwnership` интерфейса DDI вызывается со значением второго параметра `FALSE`. Поэтому драйвер больше не является владельцем политики энергопотребления, и SDV отмечает эту информацию.
- ◆ Внутри этой же библиотечной функции вызывается функция `WdfDeviceInitSetPowerPolicyEventCallbacks` интерфейса DDI. Это вызывает нарушение правила, т. к. только владелец политики энергопотребления может вызывать эту функцию интерфейса DDI.

#### Примечание

Извещение о дефекте, выводимое этим правилом, в действительности является предупреждением о возможном дефекте в драйвере. Некоторые драйверы устанавливают владение политикой энергопотребления на основе внешних данных. Например, в образце драйвера `Serial` (находится в папке `%wdk%\src\kmdf\serial`) факт владения этим драйвером политики энергопотребления определяется настройкой реестра, как определено в функции обратно-

го вызова `SerialEvtDeviceAdd` этого драйвера. Так как для таких драйверов SDV не обладает информацией о внешних данных, извещение о дефекте может быть ложноположительным в случае, если одна из трех проверяемых SDV-функций интерфейса DDI для управления энергопотреблением вызывается в пути исполнении, в котором драйвер в действительности является владельцем политики энергопотребления.

## Типы ролей функций обратного вызова KMDF для SDV

В этом разделе приводится полный список типов ролей функций обратного вызова, поддерживаемых SDV.

### Полезная информация

При присвоении имен типам ролей функций обратного вызова применяется формат, по-добрый формату имен для функций-заполнителей в WDK. Например, тип роли `EVT_WDF_DEVICE_D0_EXIT` соответствует функции-заполнителю `EvtDeviceD0Exit`. Чтобы найти справочную документацию для функции обратного вызова, удалите из имени типа роли буквы WDF и следующий за ними символ подчёркивания, после чего выполните в WDK поиск по полученной строке. Во время поиска регистр строки не имеет значения.

Далее следует список типов ролей функций обратного вызова KMDF, поддерживаемых в SDV:

```
EVT_WDF_CHILD_LIST_ADDRESS_DESCRIPTION_CLEANUP
EVT_WDF_CHILD_LIST_ADDRESS_DESCRIPTION_COPY
EVT_WDF_CHILD_LIST_ADDRESS_DESCRIPTION_DUPLICATE
EVT_WDF_CHILD_LIST_CREATE_DEVICE
EVT_WDF_CHILD_LIST_DEVICE_REENUMERATED
EVT_WDF_CHILD_LIST_IDENTIFICATION_DESCRIPTION_CLEANUP
EVT_WDF_CHILD_LIST_IDENTIFICATION_DESCRIPTION_COMPARE
EVT_WDF_CHILD_LIST_IDENTIFICATION_DESCRIPTION_COPY
EVT_WDF_CHILD_LIST_IDENTIFICATION_DESCRIPTION_DUPLICATE
EVT_WDF_CHILD_LIST_SCAN_FOR_CHILDREN
EVT_WDF_DEVICE_ARM_WAKE_FROM_SO
EVT_WDF_DEVICE_ARM_WAKE_FROM_SX
EVT_WDF_DEVICE_CONTEXT_CLEANUP
EVT_WDF_DEVICE_CONTEXT_DESTROY
EVT_WDF_DEVICE_DO_ENTRY
EVT_WDF_DEVICE_DO_ENTRY_POST_INTERRUPTS_ENABLED
EVT_WDF_DEVICE_DO_EXIT
EVT_WDF_DEVICE_DO_EXIT_PRE_INTERRUPTS_DISABLED
EVT_WDF_DEVICE_DISABLE_WAKE_AT_BUS
EVT_WDF_DEVICE_DISARM_WAKE_FROM_SO
EVT_WDF_DEVICE_DISARM_WAKE_FROM_SX
EVT_WDF_DEVICE_EJECT
EVT_WDF_DEVICE_ENABLE_WAKE_AT_BUS
EVT_WDF_DEVICE_FILE_CREATE
EVT_WDF_DEVICE_FILTER_RESOURCE_REQUIREMENTS
EVT_WDF_DEVICE_PNP_STATE_CHANGE_NOTIFICATION
EVT_WDF_DEVICE_POWER_POLICY_STATE_CHANGE_NOTIFICATION
EVT_WDF_DEVICE_POWER_STATE_CHANGE_NOTIFICATION
EVT_WDF_DEVICE_PREPARE_HARDWARE
```

EVT\_WDF\_DEVICE\_PROCESS\_QUERY\_INTERFACE\_REQUEST  
EVT\_WDF\_DEVICE\_QUERY\_REMOVE  
EVT\_WDF\_DEVICE\_QUERY\_STOP  
EVT\_WDF\_DEVICE\_RELATIONS\_QUERY  
EVT\_WDF\_DEVICE\_RELEASE\_HARDWARE  
EVT\_WDF\_DEVICE\_REMOVE\_ADDED\_RESOURCES  
EVT\_WDF\_DEVICE\_RESOURCE\_REQUIREMENTS\_QUERY  
EVT\_WDF\_DEVICE\_RESOURCES\_QUERY  
EVT\_WDF\_DEVICE\_SELF\_MANAGED\_IO\_CLEANUP  
EVT\_WDF\_DEVICE\_SELF\_MANAGED\_IO\_FLUSH  
EVT\_WDF\_DEVICE\_SELF\_MANAGED\_IO\_INIT  
EVT\_WDF\_DEVICE\_SELF\_MANAGED\_IO\_RESTART  
EVT\_WDF\_DEVICE\_SELF\_MANAGED\_IO\_SUSPEND  
EVT\_WDF\_DEVICE\_SET\_LOCK  
EVT\_WDF\_DEVICE\_SHUTDOWN\_NOTIFICATION  
EVT\_WDF\_DEVICE\_SURPRISE\_REMOVAL  
EVT\_WDF\_DEVICE\_USAGE\_NOTIFICATION  
EVT\_WDF\_DEVICE\_WAKE\_FROM\_SO\_TRIGGERED  
EVT\_WDF\_DEVICE\_WAKE\_FROM\_SX\_TRIGGERED  
EVT\_WDF\_DMA\_ENABLELER\_DISABLE  
EVT\_WDF\_DMA\_ENABLELER\_ENABLE  
EVT\_WDF\_DMA\_ENABLELER\_FILL  
EVT\_WDF\_DMA\_ENABLELER\_FLUSH  
EVT\_WDF\_DMA\_ENABLELER\_SELFMANAGED\_IO\_START  
EVT\_WDF\_DMA\_ENABLELER\_SELFMANAGED\_IO\_STOP  
EVT\_WDF\_DPC EVT\_WDF\_DRIVER\_DEVICE\_ADD  
EVT\_WDF\_DRIVER\_UNLOAD  
EVT\_WDF\_FILE\_CLEANUP  
EVT\_WDF\_FILE\_CLOSE  
EVT\_WDF\_FILE\_CONTEXT\_CLEANUP\_CALLBACK  
EVT\_WDF\_FILE\_CONTEXT\_DESTROY\_CALLBACK  
EVT\_WDF\_INTERRUPT\_DISABLE  
EVT\_WDF\_INTERRUPT\_DPC  
EVT\_WDF\_INTERRUPT\_ENABLE  
EVT\_WDF\_INTERRUPT\_ISR  
EVT\_WDF\_INTERRUPT\_SYNCHRONIZE  
EVT\_WDF\_IO\_IN\_CALLER\_CONTEXT  
EVT\_WDF\_IO\_QUEUE\_CONTEXT\_CLEANUP\_CALLBACK  
EVT\_WDF\_IO\_QUEUE\_CONTEXT\_DESTROY\_CALLBACK  
EVT\_WDF\_IO\_QUEUE\_IO\_CANCELED\_ON\_QUEUE  
EVT\_WDF\_IO\_QUEUE\_IO\_DEFAULT  
EVT\_WDF\_IO\_QUEUE\_IO\_DEVICE\_CONTROL  
EVT\_WDF\_IO\_QUEUE\_IO\_INTERNAL\_DEVICE\_CONTROL  
EVT\_WDF\_IO\_QUEUE\_IO\_READ  
EVT\_WDF\_IO\_QUEUE\_IO\_RESUME  
EVT\_WDF\_IO\_QUEUE\_IO\_STOP  
EVT\_WDF\_IO\_QUEUE\_IO\_WRITE  
EVT\_WDF\_IO\_QUEUE\_STATE  
EVT\_WDF\_IO\_TARGET\_QUERY\_REMOVE  
EVT\_WDF\_IO\_TARGET\_REMOVE\_CANCELED  
EVT\_WDF\_IO\_TARGET\_REMOVE\_COMPLETE  
EVT\_WDF\_OBJECT\_CONTEXT\_CLEANUP  
EVT\_WDF\_OBJ\_ECT\_CONTEXT\_DESTROY  
EVT\_WDF\_PROGRAM\_DMA

```
EVT_WDF_REQUEST_CANCEL
EVT_WDF_REQUEST_COMPLETION_ROUTINE
EVT_WDF_TIMER EVT_WDF_TRACE_CALLBACK
EVT_WDF_WMI_INSTANCE_EXECUTE_METHOD
EVT_WDF_WMI_INSTANCE_QUERY_INSTANCE
EVT_WDF_WMI_INSTANCE_SET_INSTANCE
EVT_WDF_WMI_INSTANCE_SET_ITEM
EVT_WDF_WMI_PROVIDER_FUNCTION_CONTROL
EVT_WDF_WORKITEM
EVT_WDFDEVICE_WDM_IRP_PREPROCESS
```

Для каждого типа роли драйвер может иметь только одну функцию обратного вызова, за исключением типов ролей, перечисленных в табл. 24.4.

**Таблица 24.4. Типы ролей, позволяющие несколько функций обратного вызова**

| Тип роли функции обратного вызова   | Максимальное количество функций |
|-------------------------------------|---------------------------------|
| EVT_WDF_DPC                         | 7                               |
| EVT_WDF_INTERRUPT_SYNCHRONIZE       | 11                              |
| EVT_WDF_TIMER                       | 6                               |
| EVT_WDF_WMI_INSTANCE_EXECUTE_METHOD | 5                               |
| EVT_WDF_WMI_INSTANCE_QUERY_INSTANCE | 5                               |
| EVT_WDF_WMI_INSTANCE_SET_INSTANCE   | 5                               |
| EVT_WDF_WMI_INSTANCE_SET_ITEM       | 5                               |

# Словарь

## ATL

См. *библиотека ATL*.

## Cab-файл (cabinet file)

Файл со сжатыми инсталляционными файлами, имеющий расширение cab.

## CLSID

См. *ID класса*.

## COM

Модель COM (Component Object Model, модель составных объектов) — платформонезависимая, распределенная система для создания двоичных программных компонентов, которые могут взаимодействовать друг с другом.

## DMA

См. *прямой доступ к памяти*.

## DMA с использованием общего буфера (common-buffer DMA)

Вид DMA, в которой драйвер выделяет область в системной памяти (общий буфер) для использования совместно с устройством DMA. Иногда называется непрерывной DMA (continuous DMA).

## GUID

Глобально уникальный идентификатор (globally unique identifier). Также, *идентификатор GUID*.

## GUID класса (class GUID)

См. *GUID класса установки*.

**GUID класса установки**

GUID, идентифицирующий класс установки устройств.

**HID**

Устройство интерфейса с пользователем (human interface device), такое как мышь или клавиатура.

**ID класса (class ID, CLSID)**

Идентификатор GUID, который уникально идентифицирует определенный объект COM. Идентификаторы CLSID требуются для объектов COM, создаваемых фабрикой класса, но являются необязательными для объектов, создаваемых по-другому.

**KMDF**

Инфраструктура для драйверов режима ядра (Kernel-Mode Driver Framework).

**MSI**

Прерывание, инициируемое сообщением (message-signaled interrupt).

**NTSTATUS**

Тип, используемый для возвращения информации о статусе многими процедурами режима ядра.

**Plug and Play**

Комбинация системного программного обеспечения, аппаратного обеспечения устройства и драйверной поддержки устройства, посредством которой компьютерная система может с минимальным или вообще отсутствующим вмешательством со стороны конечного пользователя распознавать изменения в конфигурации аппаратного обеспечения и настраиваться для работы с этими изменениями. Также называется PnP.

**SDV**

Инструмент Static Driver Verifier.

**SYS**

Расширение имени двоичных файлов драйверов режима ядра.

**UMDF**

Инфраструктура для драйверов пользовательского режима (User-Mode Driver Framework).

**Wait-блокировка (wait lock)**

Механизм синхронизации уровня PASSIVE\_LEVEL, реализуемый KMDF посредством объекта типа WDFWAITLOCK.

**WDF**

Драйверная модель WDF (Windows Driver Foundation).

**WDM**

См. *драйверная модель Windows*.

**x64**

Обозначает системы с процессорной архитектурой на основе архитектуры *x86* с расширениями для исполнения 64-битного программного обеспечения, включая процессоры AMD64 и процессоры Intel с Extended Memory 64 Technology (EM64T).

**x86**

Обозначает системы с 32-битным процессором.

**Адресное пространство ядра (kernel address space)**

Блок виртуальной памяти, выделенный для использования кодом режима ядра.

**Аннотация (annotation)**

Макрос, вставляемый в исходный код, чтобы помочь PREfast анализировать код более эффективным способом.

**Асинхронный ввод/вывод (asynchronous I/O)**

Модель ввода/вывода, в которой операции для удовлетворения запросов ввода/вывода не обязательно выполняются в первоначальной последовательности. Приложение, выдавшее запрос, может продолжать исполнение, вместо ожидания завершения этого запроса, менеджер ввода/вывода или драйвер более высокого уровня может переупорядочить поступающие запросы ввода/вывода, а драйвер самого низшего уровня может начать операцию ввода/вывода с устройством, не ожидая завершения предыдущего запроса, особенно в много-процессорных системах.

**Атомарная операция (atomic operation)**

Операция, которая должна исполниться без прерываний.

**Библиотека ATL (Active Template Library, ATL)**

Набор шаблонных классов C++, которые часто применяются для упрощения процесса создания объектов COM.

**Блокировка представления (presentation lock)**

Созданная инфраструктурой блокировка для объектов устройства и очереди, применяемая инфраструктурой для реализации области синхронизации.

**Блокировка синхронизации объекта (object synchronization lock)**

См. *блокировка представления*.

**Ввод/вывод типа NEITHER (neither I/O)**

Ни буферизованный (buffered), ни прямой (direct) ввод/вывод.

**Взаимоблокировка (deadlock)**

Ошибка исполнения, когда два потока не могут продолжать исполнение, т. к. каждый из них ожидает получения ресурса, удерживаемого другим.

**Владелец политики энергопотребления устройства (device power policy owner)**

Драйвер, управляющий политикой энергопотребления устройства, которая определяет условия перехода устройства в состояние пониженного энергопотребления и возвращение в рабочее состояние.

**Внутренний запрос IOCTL (internal device I/O control request)**

Тип запроса IOCTL, который может быть выдан только компонентом ядра.

**Внутрипроцессный объект COM (in-process COM object)**

Сервер COM, исполняющийся в контексте процесса своего клиента. Также, *объект InProc (InProc object)*.

**Вызвать исключение (raise an exception)**

Преднамеренная передача управления обработчику исключения, когда происходит исключение. Компонент режима ядра, включая любой драйвер режима ядра, не может вызвать исключение при исполнении на уровне  $\text{IRQL} \geq \text{DISPATCH\_LEVEL}$ , не вызвав при этом фатального сбоя системы.

**Вытеснение потока (thread preemption)**

Замена операционной системой потока, исполняющегося на данном процессоре, другим потоком, обычно имеющим высший приоритет потока, на этом же процессоре.

**Дескриптор (handle)**

Непрозрачный тип, посредством которого драйвер KMDF идентифицирует объект KMDF.

**Дескриптор безопасности (security descriptor)**

Структура данных, в которой хранится информации о безопасности объекта, включая владельца, группу, атрибуты защиты и контрольную информацию.

**Динамическая верификация (dynamic verification)**

Применение динамического анализа для верификации соответствия программы спецификации или протоколу.

**Динамический анализ (dynamic analysis)**

Анализ программы, осуществляемый при исполнении программы. Примерами динамического анализа являются тестирование устройства и нагружочное тестирование.

**Диспетчер (dispatcher)**

1. Системный планировщик потоков. 2. Компонент UMDF, который направляет запросы ввода/вывода получателям ввода/вывода, не являющимся частью стека устройств UMDF.

**Дополнительная установка (alternate setting)**

В интерфейсе USB, коллекция конечных точек, которые описывают функцию устройства.

**Драйвер BSD (boot-start driver)**

Драйвер, который устанавливается во время загрузки операционной системы.

**Драйвер класса (class driver)**

Драйвер, который обычно предоставляет независимую от аппаратуры поддержку для класса физических устройств и поставляется корпорацией Microsoft.

**Драйвер фильтра (filter driver)**

Драйвер, который модифицирует или отслеживает запросы ввода/вывода при их прохождении через стек устройств.

**Драйвер шины (bus driver)**

Драйвер, который перечисляет устройства, подключенные к шине.

**Драйверная модель Windows (Windows Driver Model, WDM)**

Драйверная модель, предшествующая модели WDF, поддерживающая разработку драйверов для семейства операционных систем Microsoft Windows NT, начиная с Windows 2000.

**Драйверный пакет (driver package)**

Инсталляционный пакет, включающий двоичный файл драйвера и все вспомогательные файлы для него.

**Журнал системных событий (system event log)**

Файл, содержащий журнал системных событий.

**Журнал трассировки (trace log)**

Последовательность сообщений трассировки.

**Запрос IOCTL (device I/O control request)**

Запрос ввода/вывода, не являющийся запросом ни на чтение, ни на запись и обычно выполняющий с аппаратурой какую-либо другую операцию.

**Запрос в транзите (in-flight request)**

Запрос, который был отправлен драйверу, но который еще не был завершен или переслан в другую очередь.

**Зондирование (probe)**

Проверка, что диапазон адресов памяти находится в адресном пространстве пользовательского режима и что с данным диапазоном можно выполнять операции чтения и записи в контексте текущего процесса.

**Имперсонация (impersonation)**

Способность процесса пользовательского режима исполняться с полномочиями безопасности определенного пользователя.

**Имя символьной ссылки (symbolic link name)**

Имя для устройства, которое обычно создается только старыми драйверами, которые исполняются с приложениями, использующими для доступа к устройствам имена устройств в стиле MS-DOS.

**Идентификатор IID**

Идентификатор интерфейса (interface identifier).

**Идентификатор интерфейса (interface ID)**

Идентификатор GUID, который уникально идентифицирует определенный объект COM. Независимо от того, какой объект предоставляет его, интерфейс всегда имеет один и тот же идентификатор IID.

**Идентификатор устройства (device ID, hardware ID)**

Определяемая производителем строка идентификатора устройства, которая является наиболее специфичным идентификатором, используемым утилитой Setup для соотнесения устройства с INF-файлом.

**Идентификатор экземпляра (instance ID)**

Строка идентификатора устройства, которая отличает устройство от других устройств такого же типа.

**Идентификатор экземпляра устройства (device instance ID)**

Предоставляемая системой идентификационная строка, которая однозначно идентифицирует экземпляр устройства в системе.

**Интерфейс DDI**

См. *интерфейс драйвера устройства*.

**Интерфейс USB (USB interface)**

Коллекция дополнительных установок, которые описывают варианты функции для USB-устройства. В любой момент времени можно применять только одну дополнительную установку.

**Интерфейс драйвера устройства (device driver interface, DDI)**

Коллекция системных процедур, вызываемых драйвером для взаимодействия с сервисами ядра. По существу, интерфейс DDI — это эквивалент интерфейса API для драйверов.

**Интерфейс устройства (device interface)**

Функциональность устройства, которую драйвер предоставляет приложениям или другим системным компонентам. Каждый интерфейс устройства является членом класса интерфейсов устройства, определенного системой или поставщиком.

**Исключение (exception)**

Состояние ошибки синхронного исполнения, вызванное исполнением определенной машинной команды.

**Канал (pipe)**

Конечная точка устройства USB, сконфигурированного в текущем наборе установок интерфейса.

**Класс интерфейсов устройства (device interface class)**

Группа интерфейсов, которые обычно применимы к определенному типу устройства и являются средством, с помощью которого драйверы предоставляют устройство приложениям и другим драйверам.

**Класс типа функции (function type class)**

Категория, используемая инструментом PREfast для выполнения более строгого соотнесения типов для назначений функций обратного вызова указателям на функции и для выполнения проверок, специфичных для функций определенного типа.

**Клиент COM (COM client)**

Процесс, использующий объекты COM.

**Код IOCTL**

См. код управления вводом/выводом.

**Код управления вводом/выводом (I/O control code)**

Код управления, используемый для идентификации определенной операции управления вводом/выводом устройства. Также, код *IOCTL*.

**Команда отладчика (debugger command)**

Утилита командной строки отладчика WinDbg, с помощью которой можно выполнять различные базовые отладочные операции.

**Конечная точка (endpoint)**

Получатель операции ввода, вывода или управления в устройстве USB.

**Контекст потока (thread context)**

Рабочая среда, определяемая значениями в структуре CONTEXT потока, включая права доступа и содержимое адресного пространства.

**Контекст произвольного потока (arbitrary thread context)**

Контекст потока, исполняющегося процессором в момент, когда система "одалживает" его для исполнения процедуры драйвера, например, такой как процедура ISR. Процедура драйвера не может делать никаких предположений о содержимом адресного пространства.

**Контроллер трассировки (trace controller)**

Приложение, которое активирует трассировку и соединяет поставщика трассировки с потребителем трассировки.

**Критическая область (critical region)**

Механизм блокировки, предотвращающий доставку большинства вызовов асинхронных процедур. Код, исполняющийся в критической области, выполняется на промежуточном уровне IRQL между уровнями PASSIVE\_LEVEL и APC\_LEVEL. Иногда называется критическим разделом (critical section).

**Критический раздел (critical section)**

См. *критическая область*.

**Логическое адресное пространство (logical address space)**

См. *логическое адресное пространство шины устройства*.

**Логическое адресное пространство шины устройства (device bus logical address space)**

Адресное пространство для шины устройства.

**Менеджер PnP (PnP manager)**

Подсистема ядра, управляющая операциями Plug and Play.

**Менеджер драйверов (driver manager)**

Компонент UMDF, который создает и удаляет хост-процессы драйверов, содержит информацию об их статусе, а также отвечает на сообщения от отражателя.

**Менеджер объектов (object manager)**

Подсистема ядра, управляющая объектами ядра.

**Механизм DMA "разбиение/объединение" (scatter/gather DMA)**

Вид прямого доступа к памяти, при котором осуществляется обмен данными между фрагментированными диапазонами физической памяти.

**Мьютекс (mutex)**

Объект синхронизации уровня PASSIVE\_LEVEL, который предоставляет взаимоисключающий доступ к разделяемой области памяти.

**Нарушение безопасности (security violation)**

Попытка процесса пользовательского режима получить доступ к объекту, не имея требуемых прав доступа для запрошенной операции.

**Немаскируемое прерывание (nonmaskable interrupt, NMI)**

Прерывание, которое не может быть прервано другим запросом на обслуживание. Аппаратное прерывание называется немаскируемым, если оно обходит запросы прерываний, выдаваемые программами, клавиатурой и другими устройствами, и получает более высокий приоритет, чем эти прерывания.

**Неожиданное удаление (surprise removal)**

Удаление устройства пользователем без применения Менеджера устройств или приложения для безопасного извлечения устройства.

**Нестраницный пул (non-paged pool)**

Область памяти ядра, которая всегда находится в памяти и никогда не вытесняется на диск.

**Неуправляемая энергопотреблением очередь (non-power-managed queue)**

Очередь, управление которой происходит независимо от состояния Plug and Play и энергопотребления родительского объекта устройства. Такая очередь продолжает отправлять запросы после того, как устройство переходит из рабочего состояния в состояние пониженного энергопотребления.

**Область блокировки (locking constraint)**

См. *область синхронизации*.

**Область контекста (context area)**

Определенная драйвером область в объекте WDF, в которой драйвер хранит информацию, специфичную для данного объекта.

**Область синхронизации (synchronization scope)**

Конфигурируемый механизм на основе объектов, посредством которого драйвер может указать степень параллелизма для определенных объектов обратного вызова.

**Область синхронизации очереди (queue synchronization scope)**

Механизм синхронизации WDF, в котором инфраструктура захватывает блокировку на уровне объекта очереди.

**Область синхронизации устройства (device synchronization scope)**

Механизм синхронизации WDF, в котором инфраструктура захватывает блокировку для представления объекта на уровне объекта устройства.

**Обработчик прерывания (interrupt service routine)**

Процедура, реализуемая драйвером устройства для обработки аппаратных прерываний. Также, *процедура ISR*.

**Объект FDO**

См. *объект функционального устройства*.

**Объект PDO**

См. *объект физического устройства*.

**Объект драйвера (driver object)**

Объект, представляющий драйвер.

**Объект инфраструктуры (framework object)**

Объект, управляемый WDF.

**Объект коллекции (collection object)**

Объект KMDF, содержащий цепной список (linked list) других объектов KMDF любых типов.

**Объект обратного вызова (callback object)**

Созданный драйвером объект, в котором драйвер реализует интерфейсы обратного вызова, которые требуются для обработки событий, вызываемых одним или несколькими инфраструктурными объектами.

**Объект прерывания (interrupt object)**

Объект KMDF, представляющий подключение источника аппаратного прерывания и драйверной процедуры обработки прерывания к системной таблице векторов прерываний.

**Объект таймера (timer object)**

Объект KMDF, с помощью которого драйвер может запрашивать функцию обратного вызова периодически через равные промежутки времени или один раз после истечения указанного периода времени.

**Объект устройства (device object)**

Объект устройства, представляющий участие драйвера в обработке запросов ввода/вывода определенного устройства.

**Объект устройства Down (Down device object)**

Объект устройства отражателя, который получает запросы от хост-процесса драйвера UMDF и передает их стеку устройств режима ядра.

**Объект устройства Up (Up device object)**

В отражателе, получатель запросов ввода/вывода от менеджера ввода/вывода.

**Объект устройства управления (control device object, CDO)**

1. Объект устройства, не являющийся частью стека устройств Plug and Play. 2. Когда пишется с заглавных букв, объект устройства, являющийся получателем запросов ввода/вывода между рефлектором и менеджером драйверов.

**Объект устройства фильтра (filter device object)**

Объект устройства, представляющий объект для драйвера фильтра. Также, *объект FiDO*.

**Объект физического устройства (physical device object)**

Объект устройства, представляющий объект для своего драйвера шины. Также, *объект PDO*.

**Объект функционального устройства (functional device object)**

Объект устройства, представляющий объект для функционального драйвера. Также, *объект FDO*.

**Ограничитель (guard)**

Неявное условие, накладываемые на аргумент указателя процедуры действия для целей анализа с помощью SDV.

**Останов bugcheck (bugcheck)**

Ошибка, генерируемая, когда целостность структур ядра Windows была безвозвратно нарушена. Иногда также называется *системным сбоем*.

**Отложенный вызов процедуры (deferred procedure call, DPC)**

Поставленный в очередь вызов функции режима ядра, исполняющейся на уровне IRQL = DISPATCH\_LEVEL.

**Отражатель (reflector)**

Драйвер фильтра на вершине стека устройств режима ядра, который облегчает взаимодействие между стеком устройств режима ядра и всех драйверов UMDF в системе.

**Ошибка страницы (page fault)**

Событие, когда процесс пытается обратиться к вытесненной на диск странице памяти.

**Пакет IRP**

См. *пакет запроса ввода/вывода*.

**Пакет запроса ввода/вывода (I/O request packet)**

Структура данных, с помощью которой пакет данных и связанной информации передается между менеджером ввода/вывода и компонентами стека устройств. Также, *пакет IRP*.

**Передача DMA (DMA transfer)**

Одна аппаратная операция по передаче данных из системной памяти устройству или в обратном направлении.

**Параллелизм (concurrency)**

Одновременное исполнение двух последовательностей кода.

**Параллельная диспетчеризация (parallel dispatch)**

Метод отправления запросов очередью, когда очередь посыпает драйверу запросы ввода/вывода как можно быстрее, независимо от того, обрабатывает ли драйвер в это время другой запрос.

**Повышенные привилегии (elevated privileges)**

В Windows Vista уровень привилегий, требуемый для выполнения определенных операций.

**Повышенный уровень IRQL (raised IRQL)**

Любой уровень запроса прерывания выше, чем PASSIVE\_LEVEL.

**Подсистемы ядра (kernel subsystems)**

Компоненты ядра Windows, поддерживающие базовые сервисы, такие как Plug and Play, и предоставляющие процедуры интерфейса DDI, которые позволяют драйверам режима ядра взаимодействовать с системой.

**Политика энергопотребления (power policy)**

Набор правил, определяющих, как и когда система или устройство переходит из одного состояния энергопотребления в другое.

**Получатель ввода/вывода (I/O target)**

Объект WDF, представляющий получатель запроса ввода/вывода.

**Пользовательский режим (user mode)**

Ограниченнный режим работы, в котором исполняются приложения и драйверы UMDF и в котором запрещен прямой доступ к процедурам и структурам данных ядра Windows.

**Последовательная диспетчеризация (sequential dispatch)**

Метод отправления запросов из очереди, при котором очередь доставляет драйверу запросы ввода/вывода по одному за раз, только после завершения или передачи в другую очередь предыдущего запроса.

**Последующее состояние (post-state)**

Состояние анализа PREfast сразу же после возвращения управления вызванной функцией.

**Поставщик трассировки (trace provider)**

Компонент, генерирующий сообщения трассировки.

**Потребитель трассировки (trace consumer)**

Приложение, которое форматирует и выводит на экран сообщения трассировки.

**Право доступа (access right)**

Разрешение процессу манипулировать указанным объектом определенным способом посредством вызова системного сервиса. Разные типы системных объектов поддерживают различные типы прав доступа, которые хранятся в списке управления доступом (access control list) объекта.

**Предшествующее состояние (pre-state)**

Состояние анализа PREfast непосредственно перед вызовом функции.

**Прерывание NMI**

См. *немаскируемое прерывание*.

**Прерывание потока (thread interruption)**

Механизм, посредством которого операционная система Windows заставляет текущий поток временно исполнять код на более высоком уровне IRQL.

**Приоритет (priority)**

Атрибут потока, определяющий, когда и как часто поток планируется для исполнения.

**Приоритет потока (thread priority)**

Приоритет планирования потока, в виде значений от 1 до 31 включительно. Более высокий приоритет указывается более высоким числом.

**Прерывание (interrupt)**

Извещение, посылаемое системе, что произошло что-то вне рамок обычной обработки потока, например, аппаратное событие, которое необходимо обработать как можно скорее.

**Пробуждение**

Возвращение устройства в рабочее состояние из состояния пониженного энергопотребления.

**Проверочная сборка (checked build)**

Сборка, скомпилированная с отладочными символами и содержащая специальную поддержку для отладки. Проверочные сборки применяются только для тестирования и отладки.

**Процедура DPC (DPC)**

См. *отложенный вызов процедуры*.

**Процедура ISR**

См. *обработчик прерывания*.

**Процедура завершения ввода/вывода (I/O completion routine)**

Процедура драйвера, исполняющаяся по завершению запроса ввода/вывода нижерасположенным драйвером в стеке.

**Прямой доступ к памяти (direct memory access, DMA)**

Метод обмена данными между устройством и системной памятью без участия центрального процессора.

**Рабочее состояние системы (system working state)**

Полностью рабочее состояние энергопотребления, или S0.

**Рабочий элемент (work item)**

1. Механизм (обычно применяемый процедурами с высоким уровнем IRQL) для выполнения предпочтительной обработки в системном потоке на уровне PASSIVE\_LEVEL. 2. Объект KMDF, реализующий функциональность рабочего элемента.

**Расширение отладчика (debugger extension)**

Утилита командной строки отладчика WinDbg, предоставляющая функциональность свыше функциональности, предоставляемой командами отладчика. Расширения отладчика полезны при отладке программного обеспечения специального назначения, такого как драйверы WDF.

**Регистр отображения (map register)**

Внутренняя структура, используемая при DMA, чтобы соотнести доступную устройству логическую страницу со страницей физической памяти.

**Режим ядра (kernel mode)**

Режим работы, в котором исполняется ядро операционной системы Windows и многие драйверы.

**Ручная диспетчеризация (manual dispatch)**

Способ отправки запросов из очереди, при котором драйвер извлекает запросы из очереди самостоятельно, когда ему это нужно, вызывая метод очереди.

**Свободная сборка (free build)**

Сборка для окончательного продукта. Свободная сборка меньше размером и более эффективна, чем проверочная, но ее труднее отлаживать.

**Сервер COM (COM server)**

Объект COM, предоставляющий сервисы клиентам COM.

**Сериализация (serialization)**

Управления двумя параллельными элементами одного и того же класса, например, функциями обратного вызова или запросами ввода/вывода, чтобы предотвратить их одновременное исполнение.

**Символьная ссылка (symbolic link)**

1. Экземпляр типа объекта символьной ссылки, который представляет "мягкий псевдоним" (soft alias), сопоставляющий одно имя с другим в пределах пространства имен менеджера объектов. 2. Файловый объект со специальными свойствами. Когда в пути встречается файл символьной ссылки, то вместо открытия самого файла, файловая система направляется к целевому файлу, т. е. файлу, указанному в файле символьной ссылки.

**Синхронизация (synchronization)**

Управление операциями, разделяющими данные или ресурсы, чтобы обеспечить доступ к этим разделяемым ресурсам организованным способом.

**Событие (event)**

Возникновение или завершение какого-либо действия, на которое драйверу может быть необходимым ответить, например, прибытие запроса ввода/вывода.

**Соинсталлятор (co-installer)**

Библиотека DLL, которая дополняет операции по установке устройства, выполняемые установщиком класса.

**Сообщение MSI**

Информация, передаваемая с прерыванием, инициируемым сообщением.

**Сообщение трассировки (trace message)**

Генерируемая поставщиком трассировки строка, содержащая информацию трассировки.

**Составная аннотация (composite annotation)**

Аннотация, составленная из двух или более элементарных аннотаций и других составных аннотаций.

**Состояние гибернации (hibernate state)**

Состояние энергопотребления системы S4, в котором питание системы отключено, но система может быстро восстановить рабочее состояние по информации из файла, записанного на диск перед переходом в состояние S4.

**Состояние гонок (race condition)**

Ситуация, в которой две или больше процедур пытаются получить доступ к одним и тем же данным, и результат операции зависит от последовательности, в которой выполняется доступ.

**Состояние сна (sleep state)**

Состояние энергопотребления устройства или системы иное, нежели рабочее состояние — S0 или D0.

**Состояние энергопотребления (power state)**

Уровень потребления электроэнергии системой или отдельным устройством.

**Состояние энергопотребления системы (system power state)**

Уровень потребления электропитания системой в целом. Состояния энергопотребления покрывают диапазон от S0 до S5, где S0 — это полностью рабочее состояние, а S5 — полностью отключенное состояние.

**Состояние энергопотребления устройства (device power state)**

Уровень потребления электропитания устройством. Состояние энергопотребления устройства может быть в диапазоне от D0 до D3. D0 — полностью рабочее состояние, а D3 — выключенное состояние.

**Спин-блокировка (spin lock)**

Объект синхронизации, который обеспечивает взаимоисключающий доступ на уровне IRQL = DISPATCH\_LEVEL.

**Спин-блокировка прерывания (interrupt spin lock)**

Объект синхронизации, который можно использовать при исполнении на уровне DIRQL.

**Список преобразованных ресурсов (translated resource list)**

Список, генерируемый менеджером PnP для каждого устройства, указывающий отображение каждого аппаратного ресурса устройства в текущей системе.

**Список разбиение/объединение (scatter/gather list)**

Список, содержащий один или несколько пар "базовый адрес/длина" и описывающий физические области памяти, из которых необходимо осуществлять передачу данных посредством механизма DMA "разбиение/объединение".

**Список резервных буферов (lookaside list)**

Объект KMDF, представляющий созданный драйвером и управляемый системой список буферов фиксированного размера, которые можно выделять динамическим способом.

**Список "сырых" ресурсов (raw resource list)**

Список аппаратных ресурсов, назначаемых менеджером PnP устройству. Список "сырых" ресурсов отражает физическую конструкцию устройства.

**Статический анализ (static analysis)**

Метод тестирования, при котором исходный или объектный код программы проверяется, не исполняя код, обычно с целью обнаружения ошибок в коде.

**Статическая верификация (static verification)**

Применение статического анализа для верификации соответствия программы спецификации или протоколу.

**Стек драйверов (driver stack)**

Цепочка драйверов, ассоциированных с одним или несколькими стеками устройств для предоставления поддержки работы устройств.

**Стек устройств (device stack)**

Коллекция объектов устройств и связанных драйверов, которые обрабатывают взаимодействие с определенным устройством.

**Страницочный пул (paged pool)**

Область памяти режима ядра, которая может быть вытеснена на диск и загружена обратно в память, когда необходимо.

**Структурированная обработка исключений (structured exception handling)**

Системная поддержка передачи управления обработчику исключения при возникновении определенных исключений времени исполнения.

**Таблица MDL**

См. таблица описания памяти.

**Таблица виртуальных функций (virtual function table, VTable)**

Массив указателей на реализацию каждого метода, предоставляемого интерфейсами объекта COM.

**Таблица описания памяти (memory descriptor list, MDL)**

Непрозрачная структура, определенная менеджером памяти, которая использует массив номеров фреймов физических страниц для описания страниц, содержащих диапазон адресов виртуальной памяти.

**Тег пула (pool tag)**

Четырехбайтный литерал, ассоциированный с динамически выделяемой областью страницы памяти. Тег задается драйвером при выделении памяти.

**Текущий приоритет (current priority)**

Приоритет потока в любой данный момент времени.

**Тип роли функции (function role type)**

Объявление, которое предоставляет SDV информацию о предполагаемом использовании функции обратного вызова WDF или рабочей функции WDM.

**Транзакция DMA (DMA transaction)**

Законченная операции ввода/вывода с применением DMA, например один запрос от приложения на чтение или запись.

**Узел devnode (devnode)**

Элемент в дереве устройств PnP-менеджера. Узел devnode стека устройства используется PnP-менеджером для хранения информации конфигурации и отслеживания устройства.

**Узел устройства (device node)**

См. узел devnode.

**Управляемая энергопотреблением очередь (power-managed queue)**

Очередь, управление которой происходит в зависимости от состояния Plug and Play и энергопотребления родительского объекта устройства. Такая очередь прекращает отправление запросов, когда устройство переходит из рабочего состояния в состояние пониженного энергопотребления и начинает снова отправлять их по возвращению устройства в рабочее состояние.

## Уровень DIRQL

См. уровень запроса прерывания устройства.

## Уровень IRQL

См. уровень запроса прерывания.

## Уровень запроса прерывания (interrupt request level)

Численное значение, которое Windows назначает каждому прерыванию. В случае конфликта, прерывание с высшим уровнем IRQL имеет приоритет, и его процедура обслуживания исполняется первой. Также, уровень *IRQL*.

## Уровень запроса прерывания устройства (DIRQL)

Диапазон значений уровней IRQL, ассоциированный с прерываниями устройств. Точный диапазон уровней DIRQL зависит от определенной процессорной архитектуры.

## Уровень исполнения (execution level)

Для драйверов KMDF, атрибут некоторых объектов, ограничивающий уровень IRQL, на котором инфраструктура активирует определенные функции обратного вызова объекта.

## Уровень исполнения DISPATCH\_LEVEL

Уровень IRQL, на котором исполняется код диспетчеризации потоков Windows, код для работы со страницами, а также код многих функций драйверов режима ядра.

## Уровень исполнения PASSIVE\_LEVEL (passive execution level)

Уровень исполнения, на котором не маскируются никакие прерывания. На этом уровне исполняется большая часть прикладного и системного программного обеспечения.

## Установочный класс устройств (device setup class)

Группа устройств, которые устанавливаются и конфигурируются одинаковым образом.

## Устройство RED (root-enumerated device)

Устройство, чей узел devnode является дочерним устройством корневого устройства в дереве устройств менеджера PnP.

## Фабрика класса (class factory)

Специализированный объект COM, который клиенты используют для создания экземпляра определенного объекта COM.

## Файл CAB

См. *cab-файл*.

**Файл INF**

Текстовый файл, содержащий данные по установке драйвера.

**Файл INX**

Архитектурно-независимый файл INF.

**Файл символов (symbol file)**

Файл, содержащий информацию об исполняемом образе, включая имена и адреса функций и переменных.

**Файл языка описания интерфейсов (interface definition language file)**

Файл, в котором с помощью структурированного языка описания интерфейсов определяются интерфейсы и объекты COM.

**Файловый объект (file object)**

Объект WDF, представляющий один случай использования отдельного файла и содержащий информацию для данного использования.

**Фатальная ошибка (fatal error)**

Ошибка, после которой драйвер или система не может восстановиться.

**Фильтр [в журнале PREfast]**

Опция в утилите просмотра журнала дефектов PREfast, позволяющая скрывать или показывать определенные сообщения.

**Форсаж приоритета (priority boost)**

Набор системных констант, применяющихся для повышения приоритета потока запрашивающего приложения при завершении драйвером запроса ввода/вывода. Также называется *повышением приоритета*.

**Функция обратного вызова для завершения запроса (request completion callback)**

Реализуемая драйвером функция, которая вызывается по запросу драйвера после завершения запроса ввода/вывода созданного драйвера или переданного им вниз по стеку.

**Функциональный драйвер (function driver)**

Основной драйвер устройства.

**Хост-процесс (host process)**

Процесс, называемый Wdfhost, в котором исполняется драйвер UMDF вместе со связанными компонентами пользовательского режима.

**Хранилище свойств (property store)**

Область реестра, в которой драйвер UMDF может хранить информацию о своем устройстве.

**Чисто программный драйвер (software driver)**

Драйвер, который не управляет никаким аппаратным оборудованием, ни прямым, ни косвенным (например, посредством протокола USB) образом.

**Шум (noise)**

Предупреждения, выдаваемые PREfast, не отображающие настоящие ошибки в коде. Также называются ложноположительными результатами.

**Эксклюзивное устройство**

Устройство, для которого в любой момент времени может быть открыт только один дескриптор.

**Экземпляр устройства (device instance)**

Отдельная единица аппаратного обеспечения. Например, если компания ABC изготавливает привод CD-ROM модели XYZ, и если определенная система содержит два таких привода, то тогда имеются два экземпляра устройства модели XYZ.

**Элементарная аннотация (primitive annotation)**

Аннотация, которую можно применять, как составную часть более сложной аннотации.

**Язык IDL**

Язык описания интерфейсов (interface definition language).

# Предметный указатель

## A

Accessor 568  
Active Template Library (ATL) 575, 833  
Application Verifier 669

## B

Build, утилита 596, 597, 677, 679

## C

Control device object (CDO) 143  
CheckInf 643  
ChkINF 648  
CLSID 831  
Component Object Model (COM) 18, 564, 831  
◊ клиент 566

## D

Driver Device Interface (DDI) 34  
Debugging Tools for Windows 675  
Defect Viewer 813  
DevCon 634, 638, 648  
Device Path Exerciser 649  
DIFx 638  
DIFxAPP 633  
Direct Memory Access (DMA) 18, 533, 831  
◊ абстракция 539  
◊ с использованием общего буфера 831  
Deferred Procedure Call (DPC) 46  
DPInst 633  
Driver Verifier 54, 55, 71, 393, 508, 560, 654

## E

Event Viewer 424

## F

Functional Device Object (FDO) 38, 143  
Filter Device Object (FiDO) 39, 143

## G

GUID 831

## I

I/O Manager *См. менеджер ввода/вывода*  
INF File Syntax Checker 648  
Interface Definition Language (IDL) 575  
I/O Request Packet (IRP) 40  
Interrupt Request Level (IRQL) 46  
Interrupt Service Routine (ISR) 45

## K

KernRate 649  
KMDF Verifier 55, 664  
KRVView 650

## L

Logman 423

## M

Memory Descriptor List (MDL) 44, 53  
Memory Pool Monitor (PoolMon) 651  
METHOD\_BUFFERED 44  
METHOD\_DIRECT 44  
METHOD\_NEITHER 44

## N

Nmake, утилита 602

**P**

Physical Device Object (PDO) 38, 143  
 Plug and Play 169, 832  
 Plug and Play CPU Test (PNPCPU) 651  
 Plug and Play Driver Test (Pnpdtest) 650  
 Power Management Test Tool (PwrTest)  
     651  
 PREfast 55, 66, 507, 646, 647, 707

**S**

Serialization 375  
 Skeleton 462  
 Sleep State Chooser 672  
 Specification Language for Interface Checking  
     (SLIC) 796  
 Static Driver Defect Report 826  
 Static Driver Verifier (SDV) 55, 507, 647,  
     794  
 Synchronization 375

**T**

Tracefmt 423, 427, 428  
 Tracelog 423, 427  
 Tracepdb 423  
 Tracerpt 423  
 TraceView 423, 425  
 TraceWPP 423

**U**

UMDF Verifier 667, 687  
 UuidGen 416

**V**

Virtual function table 572  
 VTable 572

**W**

Wait-блокировка 390, 832  
 WDF 61  
 WinDbg 71, 640, 675  
 Windows Driver Model (WDM) 3  
 Windows Error Reporting (WER) 669  
 Windows Management Instrumentation  
     (WMI) 41

**A**

Аннотация 728, 833  
     ◊ вложенная 754  
     ◊ входных/выходных параметров 738  
     ◊ диагностическая 763  
     ◊ для драйвера 750, 753  
     ◊ для operandов с взаимоблокировкой 781  
     ◊ для памяти 764  
     ◊ для плавающей запятой 774  
     ◊ для ресурса 766  
     ◊ для уровней IRQL 775  
     ◊ для функции в операторах \_try 764  
     ◊ класса типа функций 772  
     ◊ параметров 762  
     ◊ размера буфера 742  
     ◊ результатов функций 758  
     ◊ составление 784  
     ◊ строк 747  
     ◊ тестирование 784  
     ◊ указателя 761  
     ◊ условная 755  
     ◊ форматирующей строки 762  
 Аппаратный ресурс 514  
 Архитектура:  
     ◊ WDF 80  
     ◊ Windows 35  
     ◊ драйверов 37

**Б**

Безопасность 69  
 Библиотека шаблонных классов 575  
 Блокировка 393  
     ◊ представления 833  
     ◊ представления объекта 377  
     ◊ синхронизации объекта 834  
 Буфер ввода/вывода 245

**В**

Ввод/вывод:  
     ◊ асинхронный 833  
     ◊ буферизованный 44, 232  
     ◊ получатель См. получатель ввода/вывода  
     ◊ поток запросов 234  
     ◊ прямой 44, 232  
     ◊ самоуправляемый 296  
 Верификация драйвера 55  
 Взаимоблокировка 393, 402, 834

**Г**

Гонки 50, 373

**Д**

Дерево устройств 39  
 Директива RUN\_WPP 412  
 Достоверность параметров 70  
 Драйвер:  
     ◊ BSD 615  
     ◊ KMDF:  
         ▫ создание объекта 140  
         ▫ структура 136  
     ◊ UMDF, структура 133  
     ◊ WDF, отладка 674  
     ◊ определение 34  
     ◊ сборка 595  
     ◊ удаление 635, 636  
     ◊ установка 612, 631, 636  
     ◊ фильтра 39  
 Драйверный пакет, распространение 630

**Ж**

Журнал трассировок 406, 425

**З**

Запрос:  
     ◊ IOCTL 228  
     ◊ Plug and Play 45  
     ◊ ввода/вывода 41, 42, 64  
         ▫ отмена 292  
         ▫ приостановка 294  
         ▫ создание 319  
     ◊ на закрытие 227  
     ◊ на запись 228  
     ◊ на очистку 227  
     ◊ на создание 227, 273  
     ◊ на чтение 228  
     ◊ обработка 243  
     ◊ отмененный 292  
     ◊ приостановленный 292  
     ◊ управления устройством 228  
     ◊ управления энергопотреблением 45  
     ◊ форматирование 327

**И**

Интерфейс 568

Инфраструктура:

- ◊ KMDF 78, 88
- ◊ UMDF 76, 83

**К**

Контекст потока 495

**М**

Макрос 55  
     ◊ FAILED 67  
     ◊ NT\_SUCCESS 67  
     ◊ PAGED\_CODE() 54  
     ◊ SUCCEEDED 67  
     ◊ WDF\_DECLARE\_CONTEXT\_TYPE\_  
         WITH\_NAME 130  
     ◊ WPP\_CLEANUP 420  
 Менеджер:  
     ◊ Plug and Play (PnP) 36, 39, 169  
     ◊ ввода/вывода 36, 42  
     ◊ энергопотребления 36  
 Метод доступа 568  
 Модель:  
     ◊ KMDF, объектная 106  
     ◊ WDF 34  
         ▫ объектная 62, 96  
         ▫ ввода/вывода 41, 64  
         ▫ драйверная 34  
 Модификатор аннотации 741

**О**

Обработка ошибок 66  
 Обратный вызов:  
     ◊ деструкции 121  
     ◊ очистки ресурсов 121  
     ◊ по событию 98  
 Объект:  
     ◊ PDO 161  
     ◊ запроса ввода/вывода 244  
     ◊ коллекции 444  
     ◊ область контекста 125  
     ◊ очереди ввода/вывода 253  
     ◊ памяти 245, 432  
     ◊ прерывания 523  
     ◊ реестра 439  
     ◊ таймера 448  
     ◊ удаление 118

- ◊ устройства 37, 142
  - физического 38
  - фильтра 39
  - функционального 38
- ◊ файла для ввода/вывода 271

#### Объект KMDF:

- ◊ время жизни 114

- ◊ создание 111

- ◊ удаление 122

#### Объект UMDF:

- ◊ время жизни 114

- ◊ создание 110

- ◊ удаление 122

#### Отладка 71

##### Очередь:

- ◊ ввода/вывода 253

- ◊ неуправляемая энергопотреблением 259

- ◊ стандартная 256

- ◊ управление 260

#### Очистка ресурса 121

##### Ошибка:

- ◊ в драйвере KMDF 90

- ◊ в драйвере UMDF 86

## П

#### Память 51

- ◊ нестраничная 52

- ◊ страничная 52

- ◊ управление 51

#### Передача DMA 534

#### Подсистема ядра 36

#### Политика энергопотребления устройства 172

#### Получатель ввода/вывода 304

- ◊ обращение 310

- ◊ общий 306

- ◊ реализация в UMDF 307

- ◊ специализированный 306

- ◊ стандартный 305

- ◊ удаленный 305
  - создание 311

#### Поток 48, 494

- ◊ планирование 494

- ◊ произвольный 58

#### Потребитель трассировки 407

#### Прерывание 46, 58, 522

#### Приоритет потока 47, 494

- ◊ динамический 494

- ◊ реального времени 494

#### Программирование в режиме ядра 45

#### Процедура обслуживания прерывания 45

#### Прямой доступ к памяти 18, 533

#### Пул памяти 52

## P

#### Расширение отладчика !dma 561

#### Расширения отладчика KMDF для DMA 561

#### Регистр отображения 541

#### Реестр 436

#### Режим:

- ◊ пользовательский 36, 58

- ◊ ядра 36, 45

#### Ресурс, очистка 115

## C

#### Сборка:

- ◊ драйвера 595

- ◊ проекта 600

#### Сериализация 375, 845

#### Синхронизация 49, 64, 372, 375, 381

#### Системный сбой 56

#### Список дескрипторов:

- ◊ памяти 44

- ◊ страниц 53

#### Стек:

- ◊ драйверов 37

- ◊ устройства 37

- ◊ ядра 52

## T

#### Таблица виртуальных функций 572

#### Тестовая система 645

#### Транзакция DMA 534

#### Трассировка 71

- ◊ WPP 676

- ◊ инициализация 418

- ◊ инструменты 423

- ◊ очистка после 420

- ◊ программная 404

## Y

#### Уровень запроса на прерывание (IRQL) 46, 497

#### Устройство:

- ◊ USB 349

- ◊ запуск 183

- ◊ извлечение из разъема 190

**Ф**

Фабрика классов 583

Файл:

- ◊ Dirs 599
- ◊ Exports 474
- ◊ Make 474
- ◊ Makefile 599
- ◊ Makefile.inc 599
- ◊ Skeleton.rc 475
- ◊ Sources 473, 599
- ◊ расширение:
  - cab 831
  - pdb 679
  - sdv 809
- ◊ ресурсов 600

Функция:

- ◊ DllMain 581
- ◊ QueryInterface 589
- ◊ генерации сообщения трассировки 409
- ◊ обратного вызова 62

**Э**

Энергопотребление 170

**Я**

Ядро ОС 36

# Windows® Driver Foundation: разработка драйверов

«Любой разработчик программного обеспечения для Windows может с легкостью работать с Windows Driver Foundation, разрабатывая драйверы высокого качества в короткие сроки».

Нар Ганапати, архитектор отделения Windows корпорации Microsoft

## Начните разрабатывать надежные драйверы с помощью команды разработчиков Windows Driver Foundation.

Модель Windows Driver Foundation (WDF) позволяет разработчику создавать простой, но работоспособный драйвер, при этом большая часть обработки событий выполняется механизмом WDF. Данная книга содержит описания принципов и методик, примеры программирования и подсказки для эффективной разработки драйверов.

### Вы научитесь:

- > использовать WDF для разработки драйверов пользовательского режима и режима ядра;
- > создавать простые и работоспособные драйверы, поддерживающие Plug and Play и управление энергопотреблением;
- > эффективно управлять синхронизацией и параллельностью в коде драйвера;
- > разрабатывать код для надежной обработки ввода-вывода;
- > создавать драйверы пользовательского режима для протокольных устройств и устройств на основе последовательной шины;
- > использовать возможности USB-инфраструктур при разработке драйверов для USB-устройств;
- > разрабатывать драйверы режима ядра для устройств прямого доступа к памяти (DMA);
- > диагностировать и устранять возможные проблемы созданных драйверов с помощью инструментов для статического анализа и верификации исходного кода;
- > применять оптимальные методики тестирования, отладки и установки драйверов.

ISBN 978-5-9775-0185-9



9 785977 501859

## Об авторах

Пенни Орвик, автор многочисленных статей на тему разработки драйверов Windows, тесно сотрудничала с командой разработчиков Windows Driver Foundation с начальных стадий проекта.

Гай Смит, специализируется в тематике драйверов устройств и коде режима ядра, имеет более десяти лет опыта разработки программной документации для технологий корпорации Microsoft, включая документацию для Windows Shell, Internet Explorer и Windows Presentation Foundation.



**БХВ-Петербург**  
Санкт-Петербург,  
ул. Есенина, 5Б  
E-mail: mail@bhv.ru  
Internet: www.bhv.ru  
Тел./факс (812) 591-6243



**Русская Редакция**  
Москва,  
Шелепехинская наб., 32  
E-mail: info@rusedit.com  
Internet: www.rusedit.com  
Тел./факс (495) 638-5-638

