



**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ**  
**ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ**  
**ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**  
**“МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**  
**ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ”**

**Кафедра “Персональные компьютеры и  
сети”**



**А. В. РОЩИН**

## **Организация ввода-вывода**

**Драйверы WDM**

**Учебное пособие**

**Часть 4**



**Москва  
2011**

УДК 681.3  
ББК 32.973.26-18.2

*Рекомендовано к изданию в качестве учебного пособия  
редакционно-издательским советом МГУПИ*

Рецензенты:

Заведующий лабораторией 49 Института проблем управления РАН д.т.н., действительный член Международной академии навигации и управления движением, профессор Дорри Манучер Хабибуллаевич

Заместитель директора ФГУ ГНИИ ИТТ "Информика", к.т.н., доцент Булгаков Михаил Вячеславович

Рощин А.В. Организация ввода-вывода. Драйверы WDM. Учебное пособие. – М.: МГУПИ, 2011. – 84 с.

Настоящее учебное пособие предназначено для подготовки студентов различных специальностей, изучающих вычислительные системы различного назначения. Для студентов, обучающихся по направлению «Информатика и вычислительная техника» учебное пособие может использоваться в курсах "Организация ввода-вывода" и "Функциональное программное обеспечение встроенных систем" для самостоятельного изучения материала и подготовки к лекциям.

WDM драйверы, описываемые в данном учебном пособии, существенно облегчают программисту использование и написание драйверов устройств, а пользователю – их использование. Эти драйверы поддерживают технологию Plug and Play и обеспечивают богатые возможности управления устройствами.

В учебном пособии объясняется, место и работа драйверов в последних представителях операционных систем семейства Windows.

УДК 681.3  
ББК 32.973.26-18.2

© Рощин Алексей Васильевич 2011  
© МГУПИ 2011

Содержание	Стр.
1 Структура WDM-драйвера	4
1.1 Назначение драйвера	5
1.2 Типы драйверов	11
1.3 WDM-драйверы	12
1.4 Иерархия устройств и драйверов	13
1.5 Порядок загрузки драйверов	18
1.6 Вопросы для самопроверки	21
2 Основные структуры данных	24
2.1 Объекты драйверов	24
2.2 Объекты устройств	26
2.3 Функция <i>DriverEntry</i>	29
2.4 Функция <i>DriverUnload</i>	32
2.5 Функция <i>AddDevice</i>	33
2.6 Создание объекта устройства	33
2.7 Инициализация расширения устройства ( <i>DEVICE_EXTENSION</i> )	35
2.8 Вопросы для самопроверки	36
3 Пакеты запросов ввода-вывода <i>IRP</i>	39
3.1 Структура <i>IRP</i>	39
3.2 Стек ввода-вывода	42
3.3 Типичная модель обработки <i>IRP</i>	43
3.4 Создание синхронных <i>IRP</i>	44
3.5 Создание асинхронных <i>IRP</i>	46
3.6 Передача пакета диспетчерской функции	47
3.7 Функция <i>IoCallDriver</i>	48
3.8 Диспетчерские функции	48
3.9 Передача <i>IRP</i> вниз по стеку	51
3.10 Постановка <i>IRP</i> в очередь для последующей обработки	52
3.11 Функция <i>StartIo</i>	52
3.12 Обработчик прерывания <i>ISR</i>	53
3.13 Функция <i>DPC</i>	54
3.14 Функции завершения	55
3.15 Вызов функций завершения	58
3.16 Очереди запросов ввода-вывода	59
3.17 Объект <i>DEVQUEUE</i>	61
3.18 Отмена запросов ввода-вывода	63
3.19 Сценарии обработки <i>IRP</i>	65
3.20 Вопросы для самопроверки	78
Список использованных источников	83

## 1 Структура WDM-драйвера

*WDM (Windows Driver Model)* была разработана для стандартизации драйверов и требований к ним. WDM-драйверы используются, начиная с операционной системы Windows 98. Такой драйвер представляет собой контейнер для функций, вызываемых операционной системой. Упрощенное представление о концепции WDM-драйвера дает рисунок 1.1 [3].

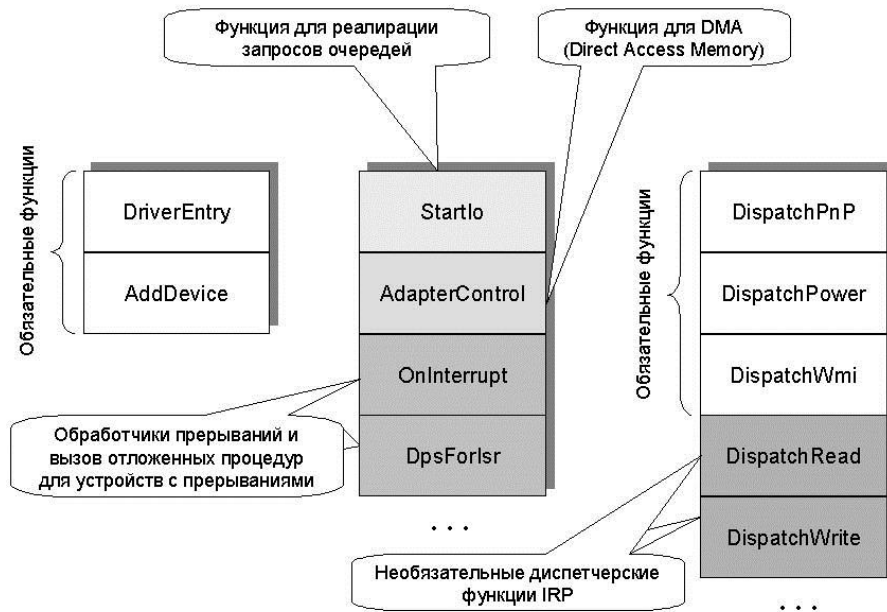


Рисунок 1.1 – Драйвер, как контейнер функций

Некоторые из указанных функций являются обязательными для каждого драйвера. Это функции *DriverEntry*, *AddDevice*, а также диспетчерские функции для некоторых типов запросов ввода-вывода *IRP*. Драйверы, использующие очереди, обычно содержат функцию *StartIo*. Драйверы, осуществляющие передачу данных по каналам прямого доступа к памяти содержат функцию *AdapterControl*. Драйверы, работающие с прерываниями, содержат обработчики прерываний (*ISR – Interrupt Service Routine*), а также функции отложенного вызова процедур (*DPC – Deferred Procedure Call*). Большая часть драйверов кроме трех необходимых диспетчерских функций содержат другие диспетчерские функции для нескольких типов *IRP*.

### *1.1 Назначение драйвера*

В Windows NT5 существует четкое разграничение двух областей в оперативной памяти и режимов процессора для исполняемого кода [1]:

- область исполняемого кода в непривилегированном режиме работы процессора (пользовательском режиме) для приложений пользователя и части компонентов операционной системы, и
- область исполняемого кода операционной системы в привилегированном режиме процессора (режиме ядра).

Под областью исполняемого кода надо понимать области загрузки (диапазон адресов) в оперативной памяти вычислительной системы. Windows 2000/XP — 32-разрядная операционная система (64-разрядную версию этой операционной системы в данном пособии не рассматриваем), и поэтому всем приложениям доступно до 4 Гбайт линейного адресного пространства. Часто в системе установлен меньший объем физической памяти, но, тем не менее, для работающих программ это незаметно. Специальные системные механизмы обеспечивают возможность виртуального присутствия 4 Гбайт памяти в системе [4]. Деление 4 Гбайт виртуального (или не виртуального, если вы можете себе это позволить) адресного пространства между пользовательскими приложениями и системными программами осуществляется поровну: первые 2 Гбайт пользовательские, остальное — системное адресное пространство.

Исполняемый код в пользовательском режиме имеет ограничения на доступ к системным ресурсам, в частности, на прямой доступ к оборудованию. Это связано с желанием обеспечить более устойчивое функционирование системы при наличии ошибок в программах пользователей. Надо учитывать, что Windows проектировалась как многозадачная и многопользовательская система, поэтому крах одного приложения не должен приводить к краху операционной системы и, следовательно, к краху других пользовательских приложений, запущенных на исполнение в этой системе. Приложения операционной системы и другие программы, исполняющиеся в режиме ядра, имеют полный доступ ко всем

ресурсам системы. Упрощенная схема архитектуры Windows NT [1,2] приведена на рисунке 1.2.

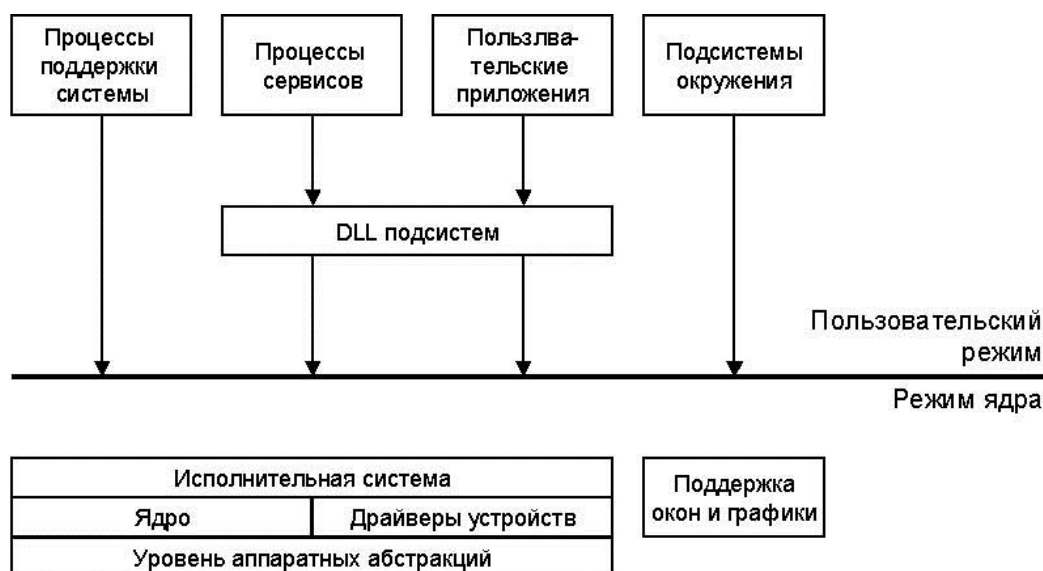


Рисунок 1.2 – Упрощенная схема архитектуры Windows NT5

Как уже отмечалось, в режиме пользователя функционируют не только прикладные программы пользователя, но и часть процессов самой операционной системы.

К компонентам операционной системы, работающим в режиме пользователя, относятся:

- некоторые процессы поддержки системы, например процесс обработки входа в систему (*Winlogon*);
- процессы Windows-сервисов. В виде сервисов оформлены как некоторые системные сервисы (например *Task Scheduler*), так и отдельные компоненты прикладных программ, например Microsoft SQL Server, а также некоторые драйверы;
- пользовательские приложения. На текущий момент они бывают шести типов: Win32, Win64 (в 64-битовой версии системы), Windows 3.1, MS-DOS, POSIX и OS/2;
- подсистемы окружения. Это часть операционной системы (программные оболочки), предоставляющая приложениям пользователя определенный для конкретной подсистемы набор функций. Windows обеспечивает работу с тремя подсистемами окружения: Win32, POSIX и OS/2. Windows

2000 поставляется с двумя подсистемами, а в Windows XP, кроме Win32, не поставляются никакие другие подсистемы окружения.

К компонентам операционной системы, работающим в режиме ядра, относятся:

- исполнительная система, обеспечивающая базовыми сервисами в части управления памятью, процессами и потоками, вводом-выводом и т.д.;
- ядро, которое содержит обобщенный набор функций операционной системы, скрывающий различия между аппаратными платформами (на разных этапах развития операционной системы Windows NT поддерживались не только процессоры Intel, но и MIPS, Alpha AXP, Motorola PowerPC). Ядро предоставляет процедуры/функции и базовые объекты, используемые исполнительной системой и драйверами для реализации структур и функций более высокого уровня. К таким функциям относятся планирование потоков, диспетчеризация прерываний, синхронизация процессов и т.д.;
- драйверы устройств;
- уровень аппаратных абстракций (*Hardware Abstraction Layer, HAL*) — набор низкоуровневых функций (около 92), обеспечивающий стандартный интерфейс взаимодействия с аппаратно-зависимыми элементами для функций, вызываемых компонентами ядра, драйверов и исполнительной системы, позволяющий абстрагироваться от того, на какой конкретно элементной базе (чипе контроллера прерывания, контроллера ПДП) реализовано выполнение доступа к шине, таймеру и т.д.;
- подсистема поддержки окон и графики.

Драйверы устройств в Windows, в отличие от DOS, для поддержки переносимости не обращаются к оборудованию напрямую, а используют функции, предоставляемые HAL. Драйверы устройств режима ядра делятся на следующие основные категории:

- драйверы файловой системы (например сетевые редиректоры и серверы).

Не стоит понимать буквально, что речь идет только о файловой системе

операционной системы. На самом деле многие физические устройства (например COM-порты) представляются в системе как файлы, и обращение к ним осуществляется посредством вызова функций, как к обычным файлам, но со специфическими параметрами. Далее уже драйверы файловой системы, получившие запрос на ввод-вывод, определяют, о каком устройстве идет речь, и вызывают соответствующие физическому устройству драйверы следующего уровня;

- драйверы с поддержкой Plug-and-Play (PnP) и ACPI (Advanced Configuration Power Management interface — усовершенствованный интерфейс управления конфигурацией и энергопотреблением);
- драйверы, не поддерживающие спецификации PnP и ACPI (например драйверы протоколов TCP/IP, IPX/SPX и т.д.), которые расширяют функциональность системы, предоставляя доступ из режима пользователя к системным сервисам и драйверам режима ядра.

В свою очередь, в каждой из категорий есть группы драйверов, которые различаются в зависимости от модели устройства и места драйверов в цепочке обработки запроса на обслуживание операций ввода-вывода.

Начиная с Windows 2000, была введена поддержка PnP и энергосберегающих технологий (ACPI), что привело к созданию модели драйверов, называемой Windows Driver Model (WDM). Здесь речь идет о линейке операционных систем NT, хотя модель драйверов WDM и была ранее реализована в Windows 98 и Windows Millennium Edition, операционная система Windows 2000 и более поздние версии линейки NT поддерживают и так называемые унаследованные драйверы (NT4), естественно, с некоторой потерей функциональности.

Модель WDM предусматривает существование трех типов драйверов:

- драйвер шины. Интересным моментом является то, что, в отличие от операционной системы NT4, Windows 2000 и выше, позволяют реализовать поддержку новых типов шин, не поддерживаемых самой операционной системой, не путем создания своего HAL (DLL), а всего



лишь добавлением своего драйвера шины. Это крайне существенно для поставщиков OEM-оборудования;

- функциональный драйвер;
- драйвер фильтра.

В рамках обобщения понятия устройства в Windows существует понятие класса устройств. Введение этого уровня абстракций сопровождается неизбежным появлением типа драйверов, отвечающих за обслуживание устройств одного класса (например CD-ROM), и драйверов, отвечающих за решение того или иного уровня взаимодействия с конкретным оборудованием. В рамках этого деления существуют драйверы:

- классов устройств;
- порт-драйверы;
- минипорт-драйверы.

Драйверы устройств в операционной системе Windows могут работать как в режиме ядра, так и в пользовательском режиме. К последним относятся:

- драйверы виртуальных устройств (VDD);
- драйверы принтеров.

Важнейшим компонентом исполнительной системы, отвечающим за связь с устройствами, является подсистема ввода-вывода. Построение подсистемы ввода-вывода, как и других компонентов операционной системы Windows призвано обеспечить максимальную устойчивость системы в целом. Поэтому в соответствии с общей доктриной разделения ответственности, связанной с режимами работы в операционной системе Windows, приложения пользователя не могут обращаться к устройствам (драйверам) напрямую, а лишь через посредников в лице диспетчеров. Некоторые компоненты подсистемы и диспетчеры, как, например, диспетчер Plug-and-Play, работают как в пользовательском режиме, так и в режиме ядра, но в целом вся подсистема и, в частности, ее главный компонент — диспетчер ввода-вывода работает в режиме ядра. В некотором промежуточном положении (с точки зрения доступности из разных режимов) оказываются inf- и cat-файлы (хранят цифровые подписи,

удостоверяющие аттестацию лаборатории Microsoft WHQL — Microsoft Windows Hardware Quality Lab) и реестр. Диспетчер ввода-вывода не только обеспечивает взаимосвязь между приложениями пользователя и драйверами устройств, но также предоставляет общий для драйверов код, используемый при обработке запросов, что существенно влияет на минимизацию кода самих драйверов. Он также обеспечивает управление буферами запросов ввода-вывода и при необходимости вызовы одним драйвером других для организации обработки запроса по цепочке. Упрощенная схема организации подсистемы ввода-вывода [2] изображена на рисунке 1.3.

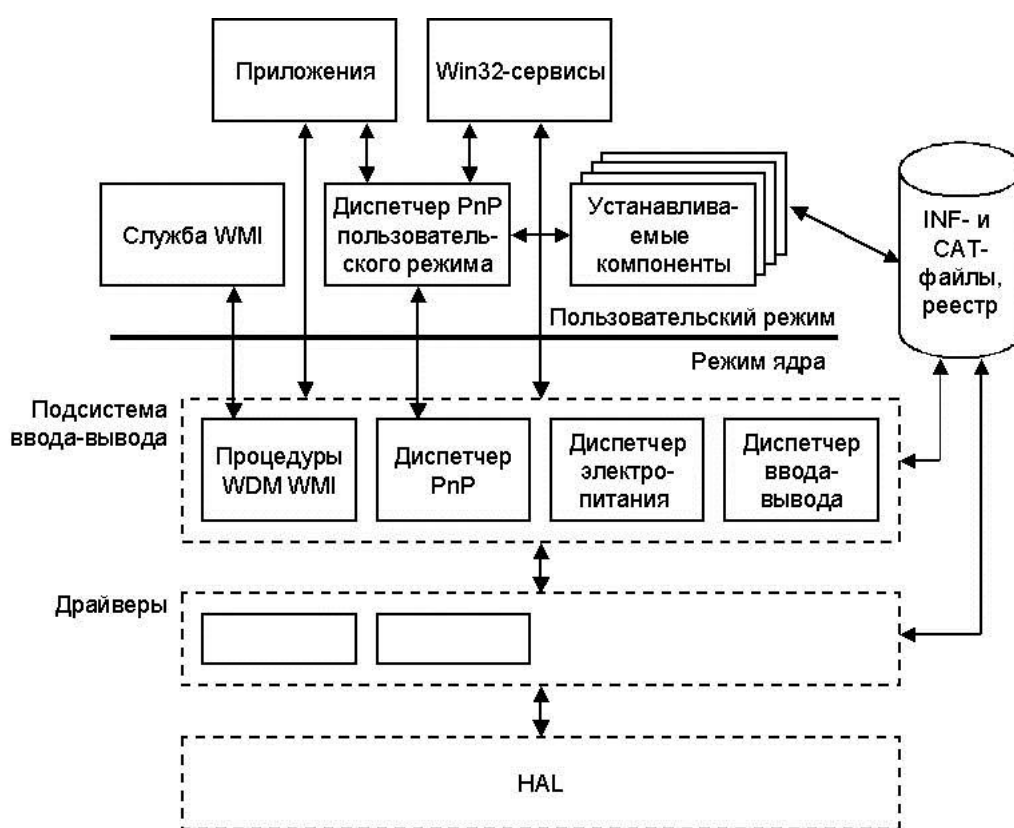


Рисунок 1.3 – Компоненты подсистемы ввода-вывода

Подсистема ввода-вывода Windows проектировалась с целью обеспечения максимальной гибкости, как с точки зрения возможности ее расширения драйверами специфических устройств, так и с учетом поддержки максимального абстрагирования устройств для прикладных приложений. Важными моментами обеспечения подобной функциональности являются возможности динамической загрузки (явной или на основе перечисления) и выгрузки драйверов, обобщенный вид формируемых структур запросов на ввод-вывод и диспетчеризация. Одним из

инструментов регистрации, запуска, останова и выгрузки драйверов служит механизм управления сервисами.

### 1.2 Типы драйверов

В Windows NT5 (2000/XP/XP Embedded) используются различные типы драйверов. Примерное представление о разнообразии их типов может дать рисунок 1.4 [3].

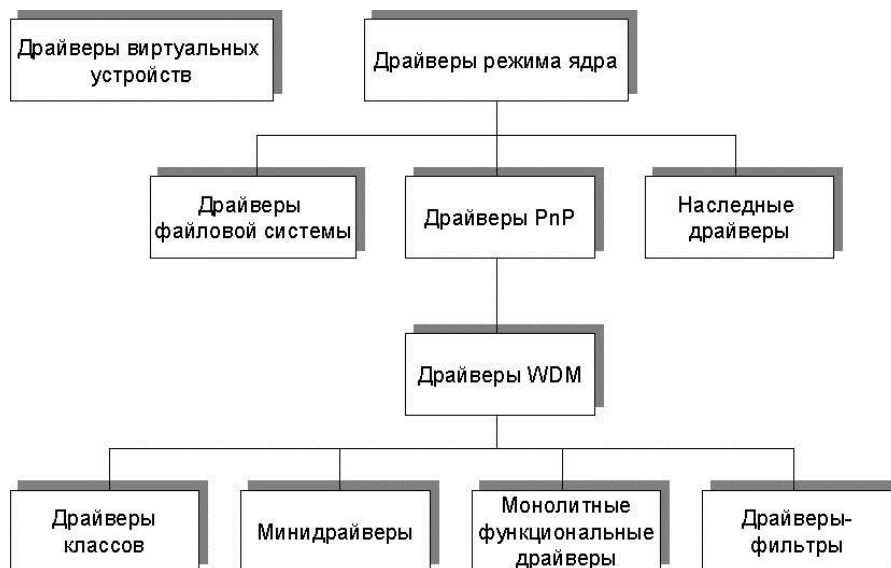


Рисунок 1.4 – Типы драйверов в Windows NT5

Драйверы виртуальных устройств (VDD – Virtual Device Driver) являются компонентами пользовательского режима, позволяющими 16-разрядным приложениям полноценно работать с оборудованием. Эти драйверы используют битовые маски ввода-вывода для перехвата обращений к портам. Таким образом они имитируют (виртуализуют) устройства для приложений, ориентированных на монопольную прямую работу с ними.

Драйверы режима ядра включают в себя несколько подвидов. Драйвер PnP – это драйвер, поддерживающий протоколы Plug and Play в системе Windows XP.

WDM-драйвером называется [3] драйвер PnP поддерживающий также протоколы управления питанием и совместимый с операционными системами Windows 98/Me/2000/XP на уровне исходных кодов.

В категорию WDM-драйверов входят также драйверы классов (для управления устройствами определенных классов), минидрайверы (предоставляющие драйверам классов специфическую поддержку устройств

конкретных производителей), монолитные функциональные драйверы (реализующие функциональную поддержку устройства) и драйверы-фильтры (перехватывающие операции ввода-вывода для конкретных устройств с целью их расширения или модификации).

Драйверы файловой системы поддерживают стандартную модель файловой системы на локальных или сетевых носителях.

Наследные драйверы устройств непосредственно управляют устройствами без помощи других драйверов. К этой категории относятся в основном драйверы ранних версий Windows NT, работающие также и в Windows XP. Главной особенностью таких драйверов является отсутствие в них поддержки стандарта Plug and Play.

### *1.3 WDM-драйверы*

При написании WDM-драйвера для конкретного устройства необходимо прежде всего решить, какой именно драйвер необходим – монолитный функциональный драйвер, драйвер-фильтр или минидрайвер. Драйверами классов операционную систему обычно обеспечивает фирма Microsoft.

Минидрайвер создается в том случае, если для устройства, которое необходимо поддерживать, у фирмы Microsoft уже есть драйвер класса. Минидрайвер будет при работе с устройством вызывать функции драйвера класса.

Драйверы-фильтры необходимы в тех случаях, когда надо лишь немного изменить реакцию существующего обобщенного драйвера от фирмы Microsoft.

Монолитные функциональные драйверы необходимы для тех устройств, которые не попадают в описанные выше категории. Такой драйвер вполне самостоятелен, и определяет все действия по управлению устройством.

Специфической особенностью драйвера, отличающей его от приложения, является отсутствие у него главного модуля, определяющего последовательность выполнения функций. Как уже говорилось, выполнение любой функции драйвера осуществляется только по запросу операционной системы.

Еще одно отличие драйвера от любого приложения заключается в том, что система не создает специального потока для выполнения кода драйвера. Функции драйвера выполняются в контексте того потока, который был активен в момент вызова системой функции. В этом случае говорят, выполнение функций драйвера происходит в контексте произвольного потока.

Однако, следует иметь в виду, что такое происходит не всегда. Драйвер может создавать собственные системные потоки, вызывая функцию *PsCreateSystemThread*.

Произвольность контекста программного потока определяет две особенности драйверов:

- драйвер не может блокировать произвольные потоки, так как это может привести к непредсказуемым последствиям,
- в произвольном потоке драйвер может создавать только асинхронные пакеты *IRP*, так как синхронные пакеты автоматически отменяются в случае завершения потока, в контексте которого они созданы.

#### 1.4 Иерархия устройств и драйверов

Прежде, чем рассматривать процессы функционирования и взаимодействия драйверов, следует рассмотреть иерархию устройств и драйверов, лежащую в основе этого взаимодействия. Концепция подобной иерархии достаточно наглядно представлена на рисунке 1.5 [3].

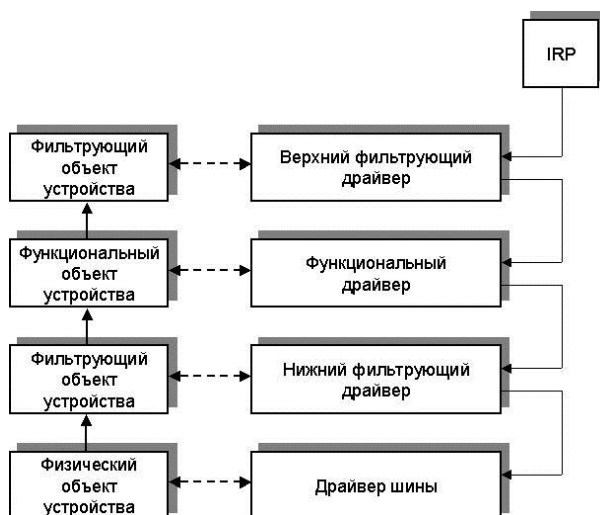


Рисунок 1.5 – Иерархия объектов устройств и драйверов в модели WDM

На рисунке 1.5 три столбца означают следующее:

- левый столбец представляет стек (направленный снизу вверх) структур ядра *DEVICE\_ОБЪЕКТ*, каждая из которых описывает специфику управления одним устройством,
- средний столбец представляет набор драйверов устройств, участвующих в управлении,
- правый столбец показывает направления передачи *IRP* (пакетов запросов ввода-вывода) между драйверами.

Модель WDM предусматривает как минимум два драйвера для каждого устройства. Один из драйверов – *функциональный драйвер* – отвечает за операции ввода-вывода, обработку прерываний, а также предоставления пользователю возможностей управления устройством. Это то, что в MS-DOS выполняет драйвер устройства.

Вторым драйвером является *драйвер шины*, отвечающий за взаимодействие оборудования с компьютером. Так, например, драйвер шины PCI (Peripheral Component Interconnect) обнаруживает карту расширения, вставленную в слот PCI, определяет требования карты к ресурсам, которые должны быть выделены для обеспечения связи с компьютером (порты ввода-вывода, области памяти), включает и выключает питание в слоте карты

Многие устройства имеют более двух драйверов. Эти дополнительные драйверы обычно называют *драйверами-фильтрами (или фильтрующими драйверами)*. Драйверы-фильтры отслеживают выполнение операций ввода-вывода функциональными драйверами или модифицируют их поведение. Верхний фильтрующий драйвер получает доступ к пакетам *IRP* до функционального драйвера, что позволяет ему поддерживать дополнительные функции, отсутствующие в функциональном драйвере. Иногда верхний драйвер-фильтр исправляет недостатки функционального драйвера.

Нижний фильтрующий драйвер получает пакеты *IRP*, которые функциональный драйвер отправляет (как он думает) драйверу шины. Его задача

также заключается в модификации, а если надо, то и в исправлении поведения функционального драйвера при выполнении операций с шиной.

Как показано на рисунке 1.5, каждый из четырех показанных драйверов устройства связан с одной из структур *DEVICE\_OBJECT* левого столбца. Эти структуры принято обозначать следующим образом:

- *PDO (Physical Device Object)* – физический объект устройства, используемый драйвером шины для представления связи устройства с шиной,
- *FDO (Function Device Object)* – функциональный объект устройства, используемый функциональным драйвером для управления функциональностью устройства,
- *FiDO (Filter Device Object)* – фильтрующий объект устройства, используемый драйвером-фильтром для хранения информации об оборудовании и о выполнении операций фильтрации.

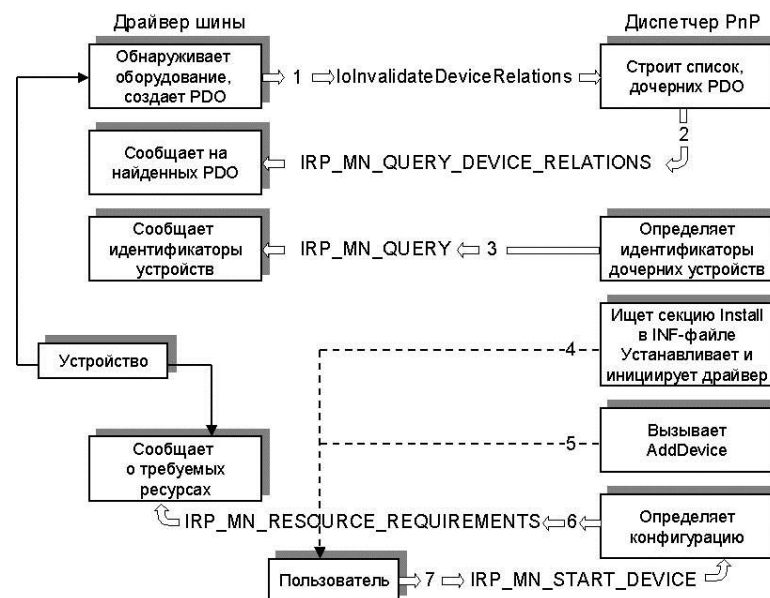


Рисунок 1.6 – Подключение устройства Plug and Play

Как уже говорилось выше, WDM-драйвер, это, прежде всего, PnP-драйвер. Любое устройство PnP обладает электронной сигнатурой, служащей для идентификации устройства. Драйвер шины Plug and Play поддерживает функцию *перечисления* (сканирования всех возможных устройств в момент запуска). Драйверы шин, поддерживающих «горячее» подключение устройств (например, USB, PCMCIA), отслеживают аппаратные сигналы, свидетельствующие о

появлении нового устройства на шине. Результатом перечисления, выполняемого в момент запуска или в момент подключения устройства, является набор объектов *PDO*, показанных на рисунке 1.6 (подробнее об объектах – в следующем разделе).

- (1) Как только драйвер шины обнаруживает факт подключения или отключения оборудования, он вызывает функцию *IoInvalidateDeviceRelations*, сообщая диспетчеру PnP список дочерних устройств.
- (2) Диспетчер PnP отправляет драйверу пакет *IRP*, который содержит основной (MJ) код функции *IRP\_MJ\_PNP* и дополнительный (MN) код функции *IRP\_MN\_QUERY\_DEVICE\_RELATIONS*, означающие, что диспетчер PnP запрашивает отношения шины (*QUERY DEVICE RELATIONS*) – (2) на рисунке. В ответ на этот запрос драйвер шины возвращает свой список объектов *PDO*, из которого диспетчер PnP может определить, какие из представленных в списке устройств еще не инициализированы.
- (3) Диспетчер PnP посылает драйверу шины пакеты *IRP* с дополнительным кодом функции *IRP\_MN\_QUERY\_ID*, запрашивающие у драйвера шины идентификаторы. Один из идентификаторов – идентификатор устройства – однозначно определяет тип конкретного устройства.
- (4) По идентификатору устройства диспетчер PnP ищет сведения об устройстве в системном реестре. Дальнейшие действия диспетчера зависят от того, имеется ли в системном реестре информация об этом конкретном устройстве. Если такой информации нет (в системном реестре отсутствует соответствующий раздел), начинает работать подсистема установки. Инструкции по установке необходимого для устройства программного обеспечения хранятся в файлах с расширением *.INF*. Каждый такой файл содержит одну или несколько команд, связывающих идентификаторы устройств с



установочными секциями в этом .INF-файле. То есть, подсистема в этом случае пытается найти .INF-файл с командой, соответствующей идентификатору устройства. Обнаружив такую команду, подсистема установки выполняет инструкции, приведенные в соответствующей секции установки. При завершении процесса установки подсистема создаст в системном реестре раздел для данного устройства.

Если же диспетчер PnP найдет в реестре сведения об устройстве, необходимость в вызове подсистемы установки отпадет, и все этапы, связанные с установкой будут пропущены.

Теперь диспетчер PnP знает, что в системе имеется устройство, за обслуживание которого отвечает конкретный драйвер. Если этот драйвер еще не загружен в виртуальную память, диспетчер PnP обращается к диспетчеру памяти с запросом на отображение драйвера, после чего система создает файловое отображение. При этом производится выборка кода драйвера и необходимых данных с использованием механизма подгрузки. Затем диспетчер памяти вызывает функцию *DriverEntry*.

Далее диспетчер PnP вызывает функцию *AddDevice*, чтобы сообщить драйверу об обнаружении нового экземпляра устройства (5) на рисунке 1.6.

Затем диспетчер PnP посылает драйверу шины пакет *IRP* с дополнительным кодом *IRP\_MN\_QUERY\_RESOURCE\_REQUIREMENTS*, который запрашивает у драйвера шины описание требований устройства к ресурсам (номер прерывания, адреса портов ввода-вывода, каналы DMA). Драйвер шины удовлетворяет этот запрос – (6) на рисунке 1.6.

Теперь диспетчер PnP готов к настройке оборудования. Если выделение ресурсов возможно, диспетчер PnP посылает драйверу пакет *IRP\_MJ\_PNP* с дополнительным кодом функции *IRP\_MN\_START\_DEVICE*. Драйвер производит настройку и подключение ресурсов, после чего устройство готово к работе.

Если устройство или драйвер (то, что называют наследным драйвером) не поддерживает стандарт Plug and Play, операционная система не в состоянии автоматически обнаружить и опознать устройство.

Для таких устройств в комплекте обычно поставляет комплектное программное обеспечение. При установке устройства пользователь в диалоговом режиме указывает системе нужную секцию нужного .INF-файла, как это показано на рисунке 1.7 (1).

Программа установки создает в реестре разделы, используемые корневым перечислителем (2). Данные, записываемые в реестр, могут включать также логическую конфигурацию и требования устройства к ресурсам (3).

В завершении подсистема установки предлагает пользователю перезагрузить систему (4). Если необходимо, при выключенном компьютере производится настройка устройства при помощи перемычек и/или переключателей в соответствии с руководством по установке.

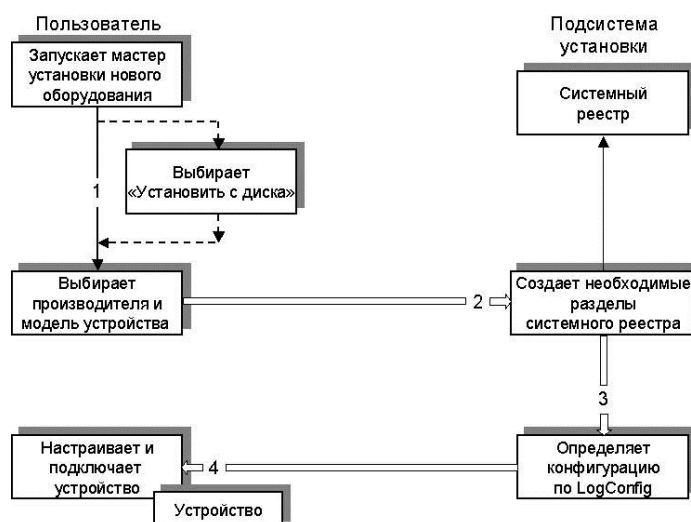


Рисунок 1.7 – Установка устройства, не поддерживающего Plug and Play

После перезагрузки системы корневой перечислитель сканирует реестр и находит устанавливаемое устройство. Далее все происходит так же, как при установке устройства, поддерживающего Plug and Play.

### 1.5 Порядок загрузки драйверов

Ранее уже было сказано, что кроме функционального драйвера устройство может обладать еще верхним и нижним драйверами-фильтрами. Информация о

драйверах-фильтрах хранятся в разделах системного реестра. В разделе устройства, содержащем информацию об экземпляре устройства, могут присутствовать параметры *UpperFilters* и *LowerFilters*, которые определяют соответственно верхние и нижние драйверы-фильтры для этого экземпляра. В реестре имеется также раздел для класса, к которому относится данное устройство. Раздел класса также может содержать параметры *UpperFilters* и *LowerFilters*. Они определяют драйверы-фильтры, загружаемые системой для каждого устройства этого класса.

Часто бывает необходимо знать, в какой последовательности система загружает драйверы. Ранее уже говорилось о том, что процесс загрузки драйвера приводит к отображению его кода на виртуальную память. Из последовательности загрузки драйверов наиболее важной является информация о порядке вызовов функций *AddDevice* драйверов. Здесь последовательность такова (рисунок 1.7) [3]:

- вызов функций *AddDevice* всех нижних драйверов-фильтров, указанных в разделе устройства в порядке их следования в параметре *LowerFilters*,
- вызов функций *AddDevice* всех нижних драйверов-фильтров, указанных в разделе класса в порядке их следования в параметре *LowerFilters*,
- вызов функции *AddDevice* функционального драйвера, определяемого параметром Service в разделе устройства,
- вызов функций *AddDevice* всех верхних драйверов-фильтров, указанных в разделе устройства в порядке их следования в параметре *UpperFilters*,
- вызов функций *AddDevice* всех верхних драйверов-фильтров, указанных в разделе класса в порядке их следования в параметре *UpperFilters*.

Для того чтобы наглядно увидеть иерархию устройств и драйверов, можно воспользоваться утилитой DevView, имеющейся на диске к [3].

На рисунке 1.9 показан результат работы утилиты на конкретном компьютере. На рисунке выбран параллельный принтер, и показана информация по объекту *PDO* (физический объект устройства), а также дополнительные данные (details).

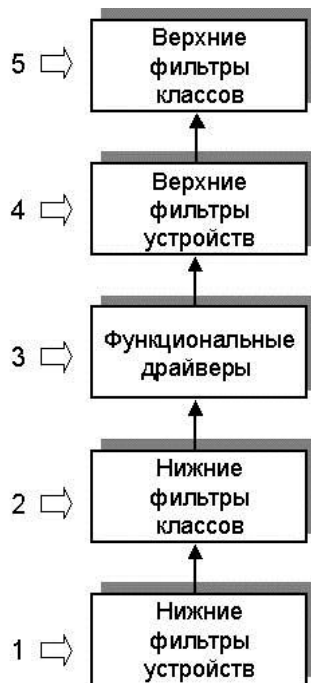


Рисунок 1.8 – Порядок вызова функций *AddDevice*

На рисунке 1.10 показана информация по объекту *FDO* (функциональный объект устройства). Из приведенных рисунков видно, что для этого устройства представлены лишь два типа объектов *PDO* и *FDO*.

Для сравнения на рисунке 1.11 показана информация для устройства – CDROM. На рисунке видно, что в устройстве представлено целых три *FiDO* – фильтрующих объекта устройства.

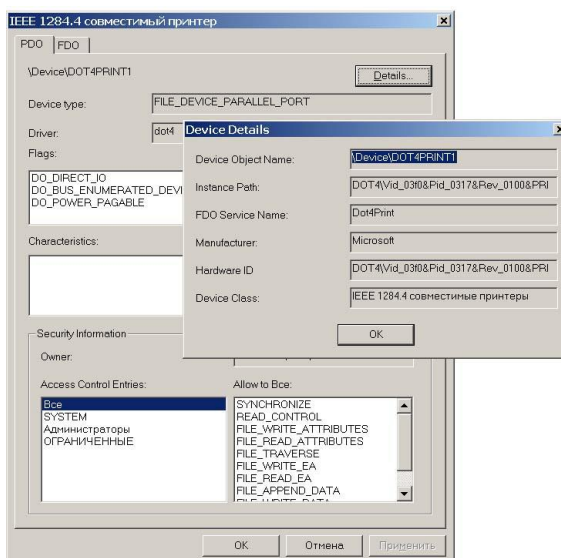


Рисунок 1.9 – Информация DevView по объекту *PDO* для параллельного принтера

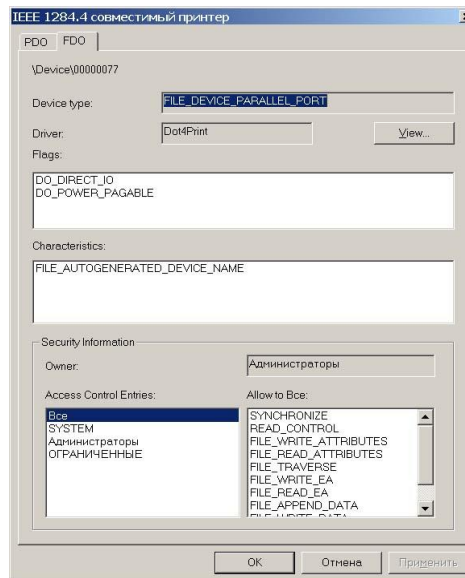


Рисунок 1.10 – Информация DevView по объекту *FDO* для параллельного принтера

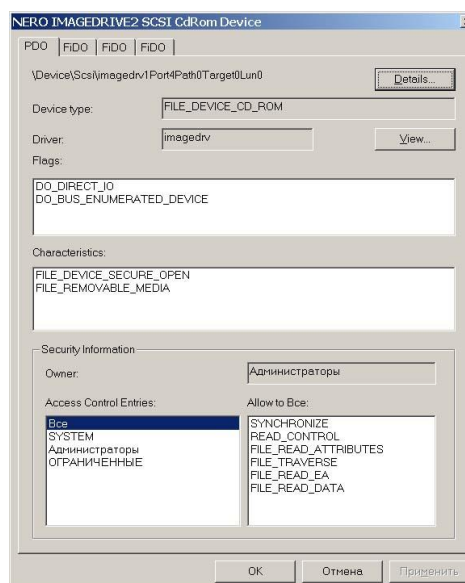


Рисунок 1.11 – Информация DevView по объекту *PDO* для устройства CDRом

### 1.6 Вопросы для самопроверки

1. Какова структура WDM-драйвера?
2. Как представить драйвер, как контейнер функций?
3. Какие функции являются обязательными в драйвере?
4. Какие необязательные функции могут присутствовать в драйвере?
5. Какие драйверы содержат обработчики прерываний?
6. Какие драйверы могут содержать функции отложенного вызова процедур?
7. Что такое область исполняемого кода драйвера?
8. Каковы особенности исполняемого кода в пользовательском режиме?
9. Каковы особенности исполняемого кода в режиме ядра?

10. Какова упрощенная архитектура Windows NT5?
11. Что такое процессы поддержки системы?
12. Что такое процессы сервисов?
13. Что такое пользовательские приложения?
14. Для чего нужны DLL подсистем?
15. Что такое подсистемы окружения?
16. Что такое исполнительная система?
17. Для чего нужно ядро?
18. Для чего нужны драйверы устройств?
19. Что такое уровень аппаратных абстракций?
20. Для чего нужна поддержка окон и графики?
21. Какие компоненты операционной системы работают в пользовательском режиме?
22. Какие компоненты операционной системы работают в режиме ядра?
23. Какие категории драйверов режима ядра Вы знаете?
24. Что такое драйверы файловой системы?
25. Что такое PnP?
26. Какие типы драйверов присутствуют в модели WDM?
27. Что такое драйвер шины?
28. Что такое функциональный драйвер?
29. Что такое драйвер фильтра?
30. Что такое подсистемы ввода-вывода?
31. Для чего нужна подсистема ввода-вывода?
32. Из каких компонентов состоит подсистемы ввода-вывода?
33. Какие типы драйверов имеются в Windows NT5?
34. Что такое минидрайвер?
35. Чем драйвер отличается от приложения?
36. В каком потоке выполняется код драйвера?
37. Может ли драйвер создавать собственные потоки?
38. Какова иерархия объектов устройств и драйверов в модели WDM?
39. Что такое физический объект устройства?
40. Что такое функциональный объект устройства?
41. Каково минимальное количество драйверов устройства?
42. Что такое функциональный драйвер?
43. Что такое драйвер шины?
44. Что такое драйвер фильтра?

45. Сколько драйверов фильтров может быть у устройства?
46. Для чего нужен верхний драйвер фильтра?
47. Для чего нужен нижний драйвер фильтра?
48. Каков порядок подключения устройства PnP?
49. Кто обнаруживает оборудование?
50. Кто создает основной физический объект устройства?
51. Кто создает список дочерних объектов устройства?
52. Для чего нужен системный реестр?
53. Кто вызывает функцию *DriverEntry*?
54. Каков порядок подключения устройства, не поддерживающего PnP?
55. Каков порядок загрузки драйверов?
56. Для чего нужен верхний драйвер фильтра?
57. Для чего нужен нижний драйвер фильтра?
58. Каков порядок вызовов функций *AddDevice*?
59. Как можно наглядно увидеть иерархию устройств и драйверов на конкретном компьютере?

## 2 Основные структуры данных

С WDM-драйверами связаны две основные структуры данных – объект драйвера и объект устройства.

Объект драйвера представляет драйвер. Он содержит указатели на функции драйвера, которые могут вызываться системой. (Здесь следует напомнить, что WDM-драйвер является контейнером функций, которые он по собственной инициативе не выполняет. Все вызовы функций драйвера выполняются операционной системой.)

Объект устройства представляет конкретный экземпляр устройства. Он содержит данные, позволяющие управлять этим устройством.

### 2.1 Объекты драйверов

Диспетчер ввода-вывода использует для представления каждого драйвера устройства объект драйвера.

Объект драйвера определяется в DDK в файле *WDM.H* следующим образом:

```
typedef struct _DRIVER_OBJECT
{
    // Объявление структуры с именем
    CSHORT Type;           // DRIVER_OBJECT, для которой
    CSHORT Size;           // объявляется тип указателя
    ...                    // PDRIVER_OBJECT и т.д.
} DRIVER_OBJECT, *PDRIVER_OBJECT; // DRIVER_OBJECT
(CSHORT – короткое целое со знаком)
```

Таблица 2.1 – Стандартные имена типов для драйверов

Имя типа	Описание
PVOID, PVOID64	Обобщенные указатели (стандартные и 64-разрядные)
NTAPI	Используется в объявлениях системных функций для использования соглашений вызова <code>__stdcall</code> на платформе i86
VOID	эквивалент <code>void</code>
CHAR, PCHAR	8-разрядный символ и указатель на него
UCHAR, PUCHAR	8-разрядный символ без знака и указатель на него
SCHAR, PSCHAR	8-разрядный символ со знаком и указатель на него
SHORT, PSHORT	16-разрядное целое со знаком и указатель на него
CSHORT	Короткое целое со знаком



Таблица 2.1 – Продолжение

USHORT, USHORT	16-разрядное целое без знака и указатель на него
LONG, PLONG	32-разрядное целое со знаком и указатель на него
ULONG, PULONG	32-разрядное целое без знака и указатель на него
WCHAR, PWSTR, PWCHAR	Символ или строка а Юникоде
PCWSTR	Указатель на константную строку в Юникоде
NTSTATUS	Код состояния (длинное целое без знака)
LARGE_INTEGER	64-разрядное целое со знаком
ULARGE_INTEGER	64-разрядное целое без знака
PSZ, PCSZ	Указатель на строку ASCIIZ с однобайтовой кодировкой или на константную строку
BOOLEAN, PBOOLEAN	TRUE или FALSE (эквивалент UCHAR)

Здесь объявляется структура с именем типа *DRIVER\_OBJECT*, для нее объявляется тип указателя *PDRIVER\_OBJECT* и назначается тэг *DRIVER\_OBJECT*.

Рассмотрим доступные поля структуры объекта драйвера.

Поле *DeviceObject* (*PDEVICE\_OBJECT*) используется для создания связанного списка объектов устройств, обслуживаемых данным драйвером. Это поле заполняет диспетчер ввода-вывода.

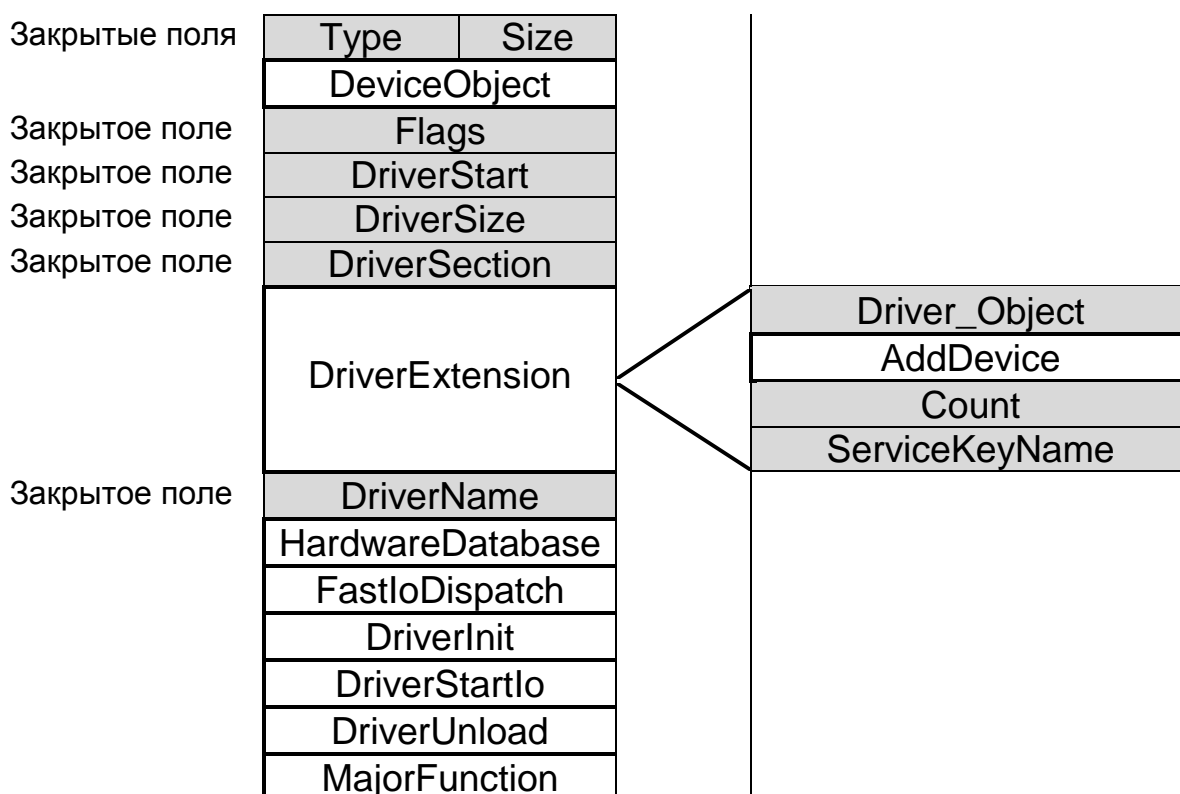


Рисунок 2.1 – Структура данных *Driver\_Object*

Поле *DriverExtension* (*PDRIVER\_EXTENSION*) указывает на вспомогательную структуру, в которой доступно только одно поле *AddDevice*

(*PDRIVER\_ADD\_DEVICE*). В *AddDevice* хранится указатель на функцию драйвера, создающую объекты устройств.

Поле *HardwareDatabase* (*PUNICODE\_EXTENSION*) хранит строку с именем раздела данного устройства в системном реестре. Имя имеет вид `\Registry\Machine\Hardware\Description\System` и определяет раздел реестра, содержащий информацию о выделенных ресурсах. Для WDM-драйверов эта информация не нужна, так как ресурсами занимается диспетчер PnP.

Во всех строковых данных режима ядра используется Юникод.

Поле *FastIoDispatch* (*PFAST\_IO\_DISPATCH*) ссылается на таблицу, которая содержит указатели на функции, экспортируемые файловой системой и сетевыми драйверами.

Поле *DriverInit* (*PDRIVER\_INIT*) содержит указатель на функцию инициализации драйвера. Функция инициализации индивидуальна для каждого драйвера.

Поле *DriverStartIo* (*PDRIVER\_STARTIO*) содержит указатель на функцию обработки запросов ввода-вывода, подготовленных диспетчером ввода-вывода.

Поле *DriverUnload* (*PDRIVER\_UNLOAD*) указывает на функцию деинициализации драйвера. (Вспомним, что выгрузкой WDM-драйверов занимается операционная система.)

Поле *MajorFunction* (*PDRIVER\_DISPATCH*) содержит таблицу указателей на функции драйвера обрабатывающие запросы ввода-вывода. Таких запросов может быть более 20 типов. Эти функции и определяют функциональность драйвера.

## 2.2 Объекты устройств

Формат объекта устройства показан на рисунке 2.2. Здесь, как и ранее, серым фоном показаны закрытые поля.

Создается такой объект вызовом функции *IoCreateDevice*. При разработке WDM-драйвера приходится часто использовать эту функцию.

Поле *DriverObject* (*PDRIVER\_OBJECT*) указывает на объект драйвера, связанный с этим устройством. Обычно это тот драйвер, который и создал данный объект устройства вызовом функции *IoCreateDevice*.

Поле *NextDevice* (*PDEVICE\_OBJECT*) указывает на следующий объект устройства, принадлежащий тому же драйверу, что и данный объект. Это поле объединяет объекты устройств драйвера в связный список, начинающийся с объекта, указанного в поле *DeviceObject* объекта драйвера.

Type	Size
ReferenceCount	
DriverObject	
NextDevice	
AttachedDevice	
CurrentIrp	
Timer	
Flags	
Characteristics	
DeviceExtension	
DeviceType	
StackSize	
...	
AlignmentRequirement	
...	

Рисунок 2.2 – Структура данных *DeviceObject*

Поле *CurrentIrp* (*PIRP*) регистрирует последний пакет *IRP*, отправленный функции *StartIo*. Это поле необходимо в том случае, если используются функции *StartPaket* и *StartNextPaket* для работы с очередями *IRP*.

Поле *Flags* (*ULONG*) содержит набор битовых флагов (таблица 2.2).

Таблица 2.2 – Флаги поля *Flags* в структуре *DEVICE\_OBJECT*

Флаг	Описание
DO_BUFFERED_IO	Операции чтения и записи используют буферизацию при обращении к данным пользовательского режима. Используется системный буфер.
DO_EXCLUSIVE	Дескриптор устройства может быть открыт только одним программным потоком.
DO_DIRECT_IO	Операции чтения и записи используют прямой доступ при работе с данными пользовательского режима. Для этого используется список дескрипторов памяти.
DO_DEVICE_INITIALIZING	Объект устройства еще не инициализирован.
DO_POWER_PAGABLE	Запрос IRP_MJ_PNP должен обрабатываться на уровне PASSIVE_LEVEL.
DO_POWER_INRUSH	Включение питания устройства сопровождается большим броском тока.

Префикс *DO\_* флагов в таблице 2.2 означает *DEVICE\_OBJECT*.

Таблица 2.3 – Флаги поля *Characteristics* в структуре *DEVICE\_OBJECT*

Флаг	Описание
FILE_REMOVABLE_MEDIA	Устройство использует сменные носители
FILE_READ_ONLY_DEVICE	Носитель поддерживает только чтение
FILE_FLOPPY_DISKETTE	Устройство – дисковод для гибких дисков
FILE_WRITE_ONCE_MEDIA	Носитель допускает только однократную запись
FILE_REMOTE_DEVICE	Доступ к устройству возможен через сетевое подключение
FILE_DEVICE_IS_MOUNTED	Физический носитель присутствует в устройстве
FILE_VIRTUAL_VOLUME	Устройство является виртуальным томом
FILE_AUTOGENERATED_DEVICE_NAME	Имя устройства автоматически генерируется диспетчером ввода-вывода
FILE_DEVICE_SECURE_OPEN	Проверка безопасности при открытии

Поле *Characteristics (ULONG)* аналогично предыдущему полю. Но его битовые флаги описывают дополнительные характеристики устройства (таблица 2.3). Диспетчер ввода-вывода устанавливает эти флаги на основании пятого аргумента функции *IoCreateDevice*.

Поле *DeviceExtension (PVOID)* указывает на структуру данных, содержащую информацию о конкретном экземпляре устройства. Структуру определяет автор драйвера.

Поле *DeviceType (DEVICE\_TYPE)* содержит константу перечислимого типа, соответствующую типу устройства. Диспетчер ввода-вывода заполняет это поле в соответствии с четвертым аргументом функции *IoCreateDevice*.

В поле *StackSize (CCHAR)* содержится количество объектов устройства от текущего до нижнего уровня *PDO*.

Поле *AlignmentRequirement (ULONG)* определяет требования к выравниванию буферов данных, используемых запросами на чтение или запись в устройство. Файл *WDM.H* содержит набор констант от *FILE\_BYTE\_ALIGNMENT* до *FILE\_512\_BYTE\_ALIGNMENT*. Значение константы определяется границей выравнивания (соответствующая степень двойки) – 1. Так *FILE\_64\_BYTE\_ALIGNMENT=0x3F (63D)*.

Далее рассмотрим основные необходимые функции драйвера.

### 2.3 Функция *DriverEntry*

Как уже говорилось выше, диспетчер PnP при обнаружении нового устройства загружает необходимые драйверы и вызывает их функции *AddDevice*.

Однако драйвер при загрузке может требовать собственной инициализации, которая выполняется при его загрузке. За эту глобальную инициализацию отвечает функция *DriverEntry*.

Ее прототип выглядит так:

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,  
    IN PUNICODE_STRING RegistryPath)  
{  
}
```

Разберем прототип по элементам.

**extern "C"** – позволяет использовать компилируемые модули C++, в котором, в отличие от C, объявлять переменные можно не только за открывающей скобкой, но и в любом другом месте. Кроме того, эта директива

обозначает использование простой генерации сигнатуры функции (в стиле языка C) при получении объектных файлов. В частности, это запрещает компилятору C++ производить "декорацию" (или "украшение") имени функции дополнительными символами при экспорте.

Ключевые слова *IN*. В *DDK IN* и *OUT* – это пустые макросы, которые служат для документирования. В параметре, перед которым стоит *IN*, функции передаются только входные данные. В параметре *OUT* функция возвращает выходные данные, а параметры *IN OUT* используются для ввода и для вывода. Таким образом, оба параметра функции являются входными.

*NTSTATUS* говорит о том, что функция возвращает значение типа *NTSTATUS*. На самом деле это обычное длинное целое (*LONG*). Лучше использовать *NTSTATUS*, так как многие функции ядра возвращают коды состояния этого типа. Список этих кодов имеется в заголовочном файле *NTSTATUS.H*.

В первом аргументе функции *DriverEntry* передается указатель на объект драйвера, представляющий ваш драйвер. Для WDM-драйвера функция *DriverEntry* должна завершить инициализацию объекта и вернуть управление.

Для драйверов других типов эта функция должна обнаружить оборудование, за которое они отвечают, создать объекты устройства, представляющие это оборудование и инициализировать их. Для WDM-драйвера всю эту работу выполняет диспетчер PnP.

Во втором аргументе функции *DriverEntry* передается указатель на структуру *PUNICODE\_STRING*, третье поле которой содержит указатель на имя соответствующего раздела реестра.

Главной задачей функции *DriverEntry* в WDM-драйвере является заполнение указателей в объекте драйвера. Эти указатели нужны для вызова функций драйвера операционной системой. Важнейшими полями-указателями объекта драйвера являются:

- *DriverUnload* – указатель на функцию деинициализации драйвера. Эту функцию вызывает диспетчер ввода-вывода перед выгрузкой драйвера.

Эта функция должна присутствовать, даже, если деинициализация не нужна. Она необходима для динамической выгрузки драйвера.

- *DriverExtension* → *AddDevice* – указатель на функцию *AddDevice* драйвера. Диспетчер PnP однократно вызывает эту функцию для каждого экземпляра оборудования, обслуживаемого драйвером.
- *DriverStartIo* – в этом поле содержится указатель на функцию *StartIo*, если драйвер использует стандартный метод формирования очередей запросов ввода-вывода.
- *MajorFunction* – диспетчер ввода-вывода помещает сюда указатели на фиктивную диспетчерскую функцию, которая сообщает о неудачной обработке запроса. Ранее уже говорилось, что драйвер должен обрабатывать какие-то *IRP*, поэтому хотя бы некоторые из указателей должны ссылаться на ваши диспетчерские функции. Драйвер обязательно должен обрабатывать запросы ввода-вывода *PnP* \_ *POWER* и *SYSTEM\_CONTROL*.

Итак, рассмотрим примерный код функции *DriverEntry*:

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = DriverUnload;           //1
    DriverObject->DriverExtension->AddDevice = AddDevice;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp; //2
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] =
        DispatchWmi;
    ...                                                    //3
    servkey.Buffer = (PWSTR) ExAllocatePool(PagePool,     //4
        RegistryPath->Length + sizeof(WCHAR));
    if (!servkey.Buffer)
        return STATUS_INSUFFICIENT_RESOURCES;
    servkey.MaximumLength = RegistryPath->Length + sizeof(WCHAR);
    RtlCopyUnicodeString(&servkey, RegistryPath);
```

```
servkey.Buffer[RegistryPath->Length/sizeof(WCHAR)] = 0;
return STATUS_SUCCESS; //5
}
```

1. Инициализация указателей на две функции, которые могут находиться в любом месте драйвера. Имена функций могут быть любыми другими, однако целесообразно давать им осмысленные имена (*DriverUnload* и *AddDevice*).
2. Каждый WDM-драйвер обязан обрабатывать два запроса ввода вывода диспетчера *PnP\_POWER* и *SYSTEM\_CONTROL*. В этом месте назначаются диспетчерские функции для обработки этих запросов. Их имена также могут быть любыми другими.
3. На месте многоточия должны формироваться указатели на другие диспетчерские функции, которые должен обрабатывать драйвер.
4. Здесь создается копия *RegistryPath* на случай, если из драйвера придется обращаться к соответствующему разделу реестра. Здесь имеется в виду, что переменная с именем *servkey* типа *UnicodeString* уже была где-то объявлена.
5. Здесь возвращается код *STATUS\_SUCCESS* (который, как всегда имеет значение 0), что означает успешное завершение функции. Если в (4) не удалось отвести память для копирования *RegistryPath*, возвращается код *STATUS\_INSUFFICIENT\_RESOURCES* (недостаточно ресурсов). Коды для неполадок различного типа содержатся в стандартном наборе.

## 2.4 Функция *DriverUnload*

Функция WDM-драйвера *DriverUnload* «убирает мусор» после начальной инициализации драйвера функцией *DriverEntry*. В нашем случае она должна освободить память, занимаемую копией *RegistryPath*.

```
VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
    RtlFreeUnicodeString(&strvkey);
}
```



Следует иметь в виду, что при завершении функции *DriverEntry* с ошибкой, система не вызывает функцию *DriverUnload*. Поэтому, если функция *DriverEntry* успела что-то натворить до выхода с ошибкой, она должна сама все исправить.

## 2.5 Функция *AddDevice*

Если драйвер обслуживает более одного физического устройства, он должен включать в себя специальную функцию *AddDevice*. Диспетчер PnP вызывает ее для каждого дополнительного устройства.

Функция выглядит примерно так:

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject,
                PDEVICE_OBJECT pdo)
{

    Return STATUS_SOMETHING; //Например, STATUS_SUCCESS
}
```

Аргумент *DriverObject* указывает на объект драйвера, инициализированный функцией *DriverEntry*. Аргумент *pdo* содержит адрес объекта физического устройства в нижней части стека.

Основные действия, выполняемые функцией *AddDevice*:

- Вызов *IoCreateDevice* для создания объекта устройства и экземпляра объекта расширения устройства.
- Регистрация одного или нескольких интерфейсов устройства для связи с приложениями. Другой способ связи с приложениями – присвоение имени объекту и создание символической ссылки.
- Инициализация объекта расширения устройства и поля *Flags* объекта устройства.
- Вызов функции *IoAttachDeviceToDeviceStack* для включения нового объекта в стек.

## 2.6 Создание объекта устройства

Для создания объекта устройства вызывается функция *IoCreateDevice*. Пример такого вызова приведен ниже:

```
PDEVICE_OBJECT fdo;
NTSTATUS status = IoCreateDevice(DriverObject,
    sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,
    FILE_DEVICE_SECURE_OPEN, FALSE, &fdo);
```

Первый аргумент функции (*DriverObject*) аналогичен первому аргументу функции *AddDevice*. Он связывает драйвер с новым объектом устройства для того, чтобы диспетчер ввода-вывода знал, кому посылать пакеты *IRP*, предназначенные для этого устройства.

Второй аргумент сообщает размер структуры расширения устройства, определенной автором драйвера (она содержит данные о конкретном экземпляре устройства). В соответствии с этим аргументом диспетчер ввода-вывода выделяет память для этой структуры и помещает указатель на нее в поле *DEVICE\_EXTENSION* объекта устройства.

Третий аргумент может содержать адрес строки *UNICODE\_STRING* с именем объекта устройства, если оно определено. В нашем случае имя не определено, поэтому значение аргумента *NULL*.

Четвертый аргумент определяет тип устройства (один из определенных в *WDM.H*), в данном случае *FILE\_DEVICE\_UNKNOWN*. Это значение может быть переопределено в разделе оборудования устройства или в разделе класса системного реестра. (Если определения записаны в обоих разделах, то раздел оборудования имеет больший приоритет.) В одном из трех мест (в аргументе функции или хотя бы в одном разделе реестра) должен быть указан правильный тип устройства. В противном случае драйвер не сможет правильно функционировать.

Пятый аргумент *FILE\_DEVICE\_SECURE\_OPEN* содержит флаги поля *Characteristics*, объекта устройства, описанные выше. Указанное значение может быть переопределено таким же образом, как и предыдущий аргумент.

Шестой аргумент (*FALSE*), позволяет задать монопольный режим обслуживания устройства. (Лучше этого не делать.) Указанное значение также может быть переопределено, как и предыдущие аргументы.

Седьмой, последний, аргумент (*&fdo*) содержит указатель на адрес созданного объекта устройства.

В случае неудачи функция возвращает соответствующий код состояния и не изменяет указатель, заданный последним аргументом. При удачном завершении функция возвращает соответствующий код и задает указатель *PDEVICE\_OBJECT*.

После успешного завершения функции можно заняться инициализацией поля расширения объекта. Если при этом возникнет ошибка, следует освободить память, занимаемую объектом устройства и вернуть соответствующий код ошибки.

Код обработки неудачи выглядит примерно так:

```
NTSTATUS status = IoCreateDevice(...);
if (!NT_SUCCESS(status))
    return status;
...
if (<какая-то еще ошибка>)
{
    IoDeleteDevice(fdo);
    return status;
}
```

## 2.7 Инициализация расширения устройства (*DEVICE\_EXTENSION*)

Содержимое расширения устройства зависит от особенностей устройства и способов его программирования. Заполнение этой структуры и работа с ней возложена полностью на автора драйвера.

В большинстве драйверов в расширение устройства включены следующие атрибуты:

```
typedef struct _DEVICE_EXTENSION {           //1
    PDEVICE_OBJECT DeviceObject;             //2
    PDEVICE_OBJECT LowerDeviceObject;        //3
    PDEVICE_OBJECT Pdo;                      //4
    UNICODE_STRING ifname;                   //5
    IO_REMOVE_LOCK RemoveLock;               //6
```

```

DEVSTATE devstate;                //7
DEVSTATE prevstate;
DEVICE_POWER_STATE devpower;
SYSTEM_POWER_STATE syspower;
DEVICE_CAPABILITIES devcaps;      //8
...
} DEVICE_EXTENSION, *DEVICE_EXTENSION;

```

Здесь (1), как и ранее приведено такое объявление структуры, которое используется в DDK. То есть, расширение устройства определено, как структура с именем *DEVICE\_EXTENSION*, типом указателя *PDEVICE\_EXTENSION* и тэгом *\_DEVICE\_EXTENSION*.

Как уже было сказано, на структуру расширения устройства указывает содержимое поля *DeviceExtension* в объекте устройства (2). Однако часто бывает необходимо найти объект устройства по структуре расширения устройства, которая подробно описывает конкретный его экземпляр. Именно для этого в расширение устройства включается указатель *DeviceObject*.

Поле *LowerDeviceObject* (3) служит для сохранения адреса нижележащего объекта устройства. Необходимость в этом возникает при вызове функции *IoAttachDeviceToDeviceStack*.

Поле *Pdo* (4) содержит адрес объекта устройства, который бывает необходим некоторым служебным функциям.

Поле *ifname* служит для хранения имени интерфейса устройства (типа *UNICODE\_STRING*), необходимое для обращению к устройству.

В поле (6) определяется объект блокировки устройства *IO\_REMOVE\_LOCK*, используемый для безопасного отключения устройства. Инициализацией этого объекта занимается функция *AddDevice*.

В поле (7) включены переменные для отслеживания текущего состояния Plug and Play и энергопотребления устройства. Предполагается, что тип *DEVSTATE* уже где-то объявлен.

Поле (8) связано с дополнительными возможностями управления питанием.

## 2.8. Вопросы для самопроверки

1. Что представляет объект драйвера?
2. Что представляет объект устройства?
3. Как объект драйвера определен в DDK?
4. Каковы стандартные имена типов для драйверов?
5. Что такое CHAR?
6. Что такое UCHAR?
7. Что такое LONG?
8. Что такое ULONG?
9. Что такое WCHAR?
10. Что такое PWCHAR?
11. Что такое PWSTR?
12. Что такое PCWSTR?
13. Что такое PSZ?
14. Для чего служит поле DeviceObject в объекте драйвера?
15. Для чего служит поле DriverExtension в объекте драйвера?
16. Для чего служит поле HardwareDatabase в объекте драйвера?
17. Для чего служит поле FastIoDispatch в объекте драйвера?
18. Для чего служит поле DriverInit в объекте драйвера?
19. Для чего служит поле DriverStartIo в объекте драйвера?
20. Для чего служит поле DriverUnload в объекте драйвера?
21. Для чего служит поле MajorFunction в объекте драйвера?
22. Для чего служит поле AddDevice в объекте драйвера?
23. Какая функция создает объект устройства?
24. Для чего служит поле DriverObject в объекте устройства?
25. Для чего служит поле NextDevice в объекте устройства?
26. Для чего служит поле CurrentIrp в объекте устройства?
27. Для чего служит поле Flags в объекте устройства?
28. Для чего служит поле Characteristics в объекте устройства?
29. Для чего служит поле DeviceExtension в объекте устройства?
30. Для чего служит поле DeviceType в объекте устройства?
31. Для чего служит поле StackSize в объекте устройства?
32. Для чего служит поле AlignmentRequirement в объекте устройства?
33. Какие флаги присутствуют в поле Flags в объекте устройства?
34. Какие флаги присутствуют в поле Characteristics в объекте устройства?
35. Для чего нужна функция DriverEntry?
36. Когда выполняется функция DriverEntry?

37. Что означает префикс extern “C” в функции DriverEntry?
38. Что означают ключевые слова IN и OUT?
39. Что такое NTSTATUS?
40. Чем отличается функция DriverEntry для WDM драйвера?
41. Какие поля-указатели должна заполнить функция DriverEntry?
42. Для чего нужно поле DriverUnload?
43. Для чего нужно поле DriverExtension->AddDevice?
44. Для чего нужно поле DriverStartIo?
45. Какие два запроса ввода вывода обязан обрабатывать каждый WDM драйвер?
46. Что делает функция DriverUnload?
47. В каком случае операционная система может не вызвать функцию DriverUnload?
48. Что делает функция AddDevice?
49. Кто вызывает функцию AddDevice?
50. Для чего нужен аргумент DriverObject в функции AddDevice?
51. Для чего нужен аргумент pdo в функции AddDevice?
52. Какие действия выполняет функция AddDevice?
53. Кто вызывает функцию IoCreateDevice?
54. Что делает функция IoCreateDevice?
55. Для чего нужна регистрация интерфейса устройства?
56. Для чего нужна инициализация объекта расширения устройства?
57. Для чего нужна инициализация поля Flags объекта устройства?
58. Кто занимается инициализацией объекта расширения устройства?
59. Кто занимается инициализацией поля Flags объекта устройства?
60. Для чего вызывается функция IoAttachDeviceToDeviceStack?
61. Кто вызывает функцию IoAttachDeviceToDeviceStack?
62. Для чего вызывается функция IoCreateDevice?
63. Каков тип объекта устройства?
64. Для чего нужен первый аргумент функции IoCreateDevice?
65. Для чего нужен второй аргумент функции IoCreateDevice?
66. Для чего нужен третий аргумент функции IoCreateDevice?
67. Для чего нужен четвертый аргумент функции IoCreateDevice?
68. Для чего нужен пятый аргумент функции IoCreateDevice?
69. Каковы приоритеты в определении типа устройства?
70. Для чего нужен шестой аргумент функции IoCreateDevice?
71. Для чего нужен седьмой аргумент функции IoCreateDevice?

### 3 Пакеты запросов ввода-вывода *IRP*

При обработке запросов ввода-вывода наиболее важными являются две структуры данных. Сам пакет и стек ввода-вывода.

#### 3.1 Структура *IRP*

Структура данных *IRP* показана на рисунке 3.1. Как и ранее закрытые поля отмечены серым фоном.

Type		Size	
MdlAddress			
Flags			
AssociatedIrp			
ThreadListEntry			
IoStatus			
RequestorMode	PendingReturned	StackCount	CurrentLocation
Cancel	Cancellrql	ApcEnvironment	AllocftionFlags
Userlosb			
UserEvent			
Overlay			
CancelRoutine			
UserBuffer			
Tail			

Рисунок 3.1 – Структура данных *IRP*

Поле *MdlAddress* (*PMDL*) содержит адрес таблицы дескрипторов памяти (*Memory Descriptor List*). В этой таблице описан буфер пользовательского режима, связанный с данным запросом. *MDL* создается диспетчером ввода-вывода.

Для запросов чтения и записи (*IRP\_MJ\_READ* и *IRP\_MJ\_WRITE*), если флаги объекта верхнего устройства указывают режим *DO\_DIRECT\_IO*.

Для запроса *IRP\_MJ\_DEVICE\_CONTROL* *MDL* создается при наличии в управляющем коде флагов *METHOD\_IN\_DIRECT* или *METHOD\_OUT\_DIRECT*.

Таблица описывает виртуальный буфер пользовательского режима и содержит физические адреса заблокированных страниц с этим буфером.

Поле *Flags* (*ULONG*) содержит флаги, которые драйвер устройства может читать, но не может изменять.

Поле *AssociatedIrp* объединяет три возможных указателя. Указатель для стандартного WDM-драйвера называется *AssociatedIrp.SystemBuffer*. Он содержит адрес системного буфера в непереключаемой памяти режима ядра.

Диспетчер ввода-вывода создает этот буфер для запросов чтения и записи (*IRP\_MJ\_READ* и *IRP\_MJ\_WRITE*), если флаги объекта верхнего устройства указывают режим *DO\_DIRECT\_IO*.

Для запроса *IRP\_MJ\_DEVICE\_CONTROL* диспетчер ввода-вывода создает этот буфер при наличии в управляющем коде флагов *METHOD\_IN\_DIRECT* или *METHOD\_OUT\_DIRECT*.

Диспетчер ввода-вывода копирует в этот буфер данные, отправленные драйверу кодом пользовательского режима при создании *IRP*. Это происходит при запросах записи и управления устройством (*IRP\_MJ\_DEVICE\_CONTROL*).

При запросах чтения этот буфер заполняется драйвером устройства. Позднее диспетчер ввода-вывода копирует эти данные в буфер пользовательского режима.

Поле *IoStatus* (*IO\_STATUS\_BLOCK*) содержит структуру с двумя полями, которые заполняет драйвер при завершении обработки запроса ввода-вывода.

Поле *IoStatus.Status* содержит код *NTSTATUS*, а поле *IoStatus.Information* дополнительные данные, зависящие от типа *IRP* и статуса завершения. Для запросов чтения при удачном его завершении в этом поле может указываться общее количество переданных байтов. Некоторые запросы PnP передают в этом поле указатель на структуру, которая и является ответом на запрос.

Поле *RequestorMode* содержит одну из констант перечисленного типа { *UserMode* или *KernelMode* }. Значение зависит от источника запроса.

Поле *PendingReturned* (*BOOLEAN*) показывает, вернула ли диспетчерская функция нижнего уровня код *STATUS\_PENDING*.

Поле *Cancel* (*BOOLEAN*) имеет значение *TRUE*, если для отмены запроса была вызвана функция *IOCancelIrp*, или *FALSE* в противном случае.



Поле *CancelIrql* (*KIRQL* – тип целого числа, содержащего значение *IRQL*) содержит уровень запроса прерывания (*IRQL*), на котором была захвачена спин-блокировка отмены.

Поле *CancelRoutine* (*PDRIVER\_CANCEL*) содержит адрес функции отмены *IRP* данного драйвера.

Поле *UserBuffer* (*PVOID*) содержит виртуальный адрес пользовательского режима для выходного буфера запроса *IRP\_MJ\_DEVICE\_CONTROL*, если для него задан режим *METHOD\_NEITHER*.

Поле *Taile* содержит объединение полей (рисунок 3.2).

Поле *Taile.Overlay* также содержит объединение альтернатив:

*Taile.Overlay.DeviceQueueEntry* (*KDEVICE\_QUEUE\_ENTRY*) и

*Taile.Overlay.DriverContext* (*PVOID[4]*).

Диспетчер ввода-вывода использует поле *Taile.Overlay.DeviceQueueEntry* для организации стандартных очередей.

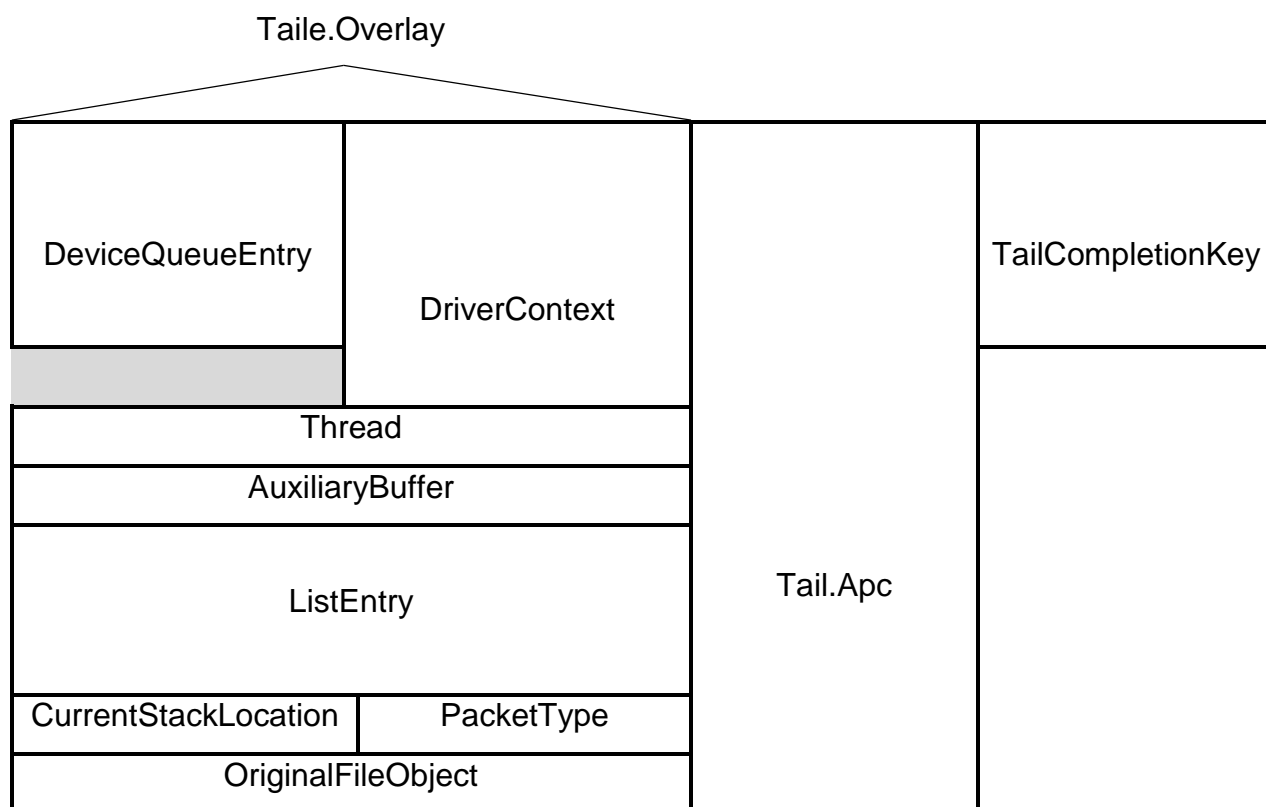


Рисунок 3.2 – Объединение *Tail* в *IRP*

Последний элемент массива *Taile.Overlay.DriverContext* используют функции *IoCsqXxx* (функции отмены безопасной очереди). Если драйвер является

владельцем *IRP*, и пакет не находится в очереди, использующей этот элемент, все четыре указателя в массиве могут быть использованы произвольно.

Поле *Taile.Overlay.ListEntry (LIST\_ENTRY)* может использоваться для организации частных очередей.

### 3.2 Стек ввода-вывода

При создании *IRP* диспетчер ввода-вывода создает также массив структур *IO\_STACK\_LOCATION*. В стеке содержится столько таких структур, сколько драйверов будет обрабатывать данный *IRP* (рисунок 3.3).

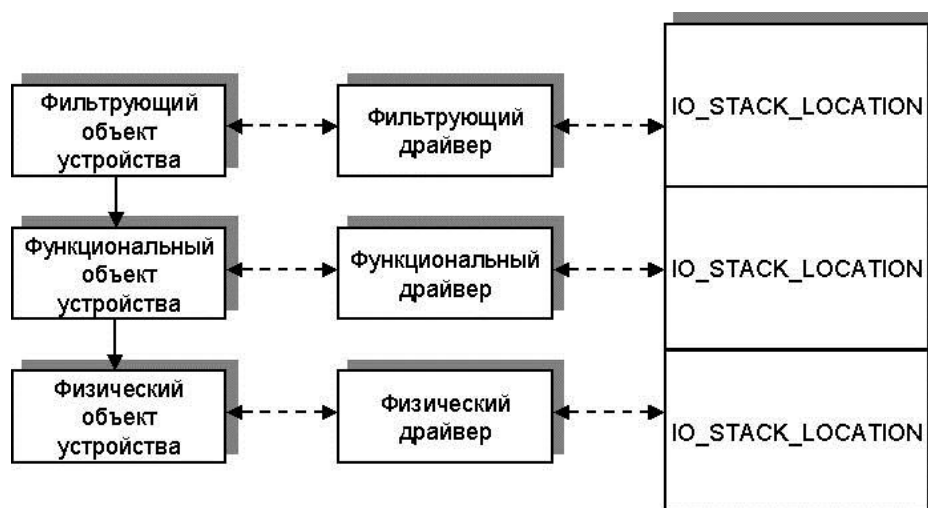


Рисунок 3.3 – Соответствие между драйверами и стеком ввода-вывода  
Структура каждого элемента стека показана на рисунке 3.4.

MajorFunction	MinorFunction	Flags	Control
Parameters			
DeviceObject			
FileObject			
CompletiouvRoutine			
Context			

Рисунок 3.4 – Структура элемента стека ввода-вывода

Поле *MajorFunction (UCHAR)* содержит основной код (значение) функции, связанной с *IRP*. Этот код, например, *IRP\_MJ\_READ* соответствует одному из элементов таблицы функций *MajorFunction* объекта драйвера.

Поле *MinorFunction* (*UCHAR*) содержит дополнительный код функции, который уточняет смысл пакетов *IRP*, если это необходимо. Например, запросы *IRP\_MJ\_PNP* подразделяются на подтипы при помощи дополнительных кодов функций, например, *IRP\_MN\_START\_DEVICE*.

Поле *Flags* содержит флаги обработки, определенные для выполняемой функции ввода/вывода. Это поле важно для драйверов файловых систем.

Поле *Control* является набором флагов, которые устанавливаются и читаются диспетчером ввода-вывода, указывая, как надо обработать данный пакет *IRP*. Например, в этом поле с помощью обращения драйвера к функции *IoMarkIrpPending* может быть установлен бит *SL\_PENDING*, указывающий диспетчеру ввода-вывода, что завершение обработки пакета *IRP* отложено.

Поле *Parameters* является объединением подструктур для каждого типа запросов, сопровождающихся заданием параметров. Например, подструктура *Create* нужна для запросов *IRP\_MJ\_CREATE*, подструктура *Read* нужна для запросов *IRP\_MJ\_READ* и т. д.

Поле *DeviceObject* (*PDEVICE\_OBJECT*) указывает на объект устройства, соответствующего данной позиции стека. Для заполнения поля вызывается функция *IoCallDriver*.

Поле *FileObject* (*PFILE\_OBJECT*) содержит адрес файла режима ядра, которому направляется *IRP*. Этот указатель часто используется для отмены всех пакетов *IRP* при закрытии соответствующего файла.

Поле *CompletionRoutine* (*PIO\_COMPLETION\_ROUTINE*) содержит адрес функции завершения ввода-вывода. Эта функция устанавливается вышестоящим (в стеке) драйвером.

Поле *Context* (*PVOID*) содержит то, что передается в качестве аргумента функции завершения.

### 3.3 Типичная модель обработки *IRP*

Типичная модель обработки *IRP* показана на рисунке 3.5. На рисунке показана смена владельца *IRP* на каждом этапе его жизни. Не каждый тип пакета

проходит весь показанный путь. Некоторые его фазы могут изменяться или исключаться.

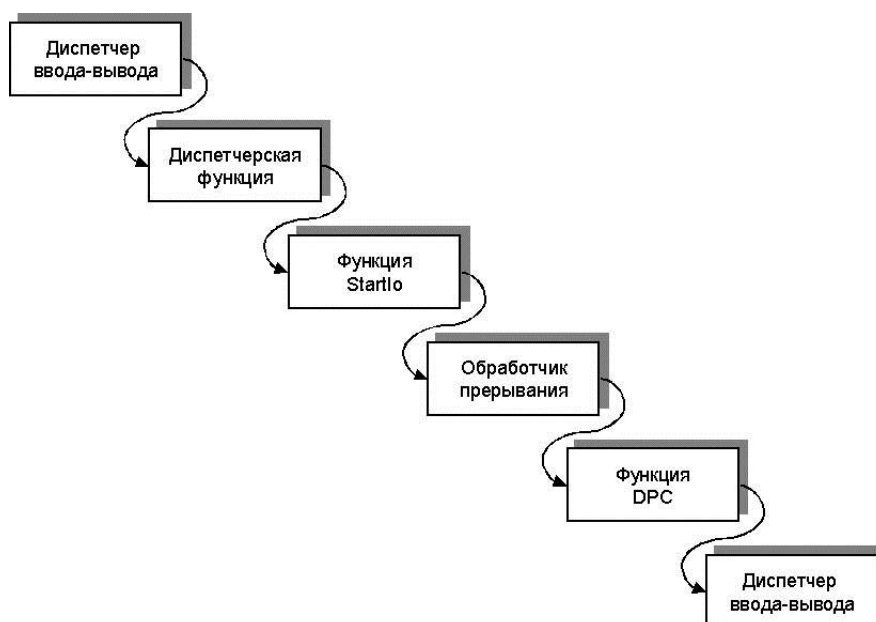


Рисунок 3.5 – Типичная модель обработки *IRP*

Жизненный цикл *IRP* начинается с его созданием диспетчером ввода-вывода. Для создания *IRP* используются четыре функции:

Функция *IoBuildAsynchronousFsdRequest* (*Fsd* – *File System Driver*) создает *IRP*, завершения которого вы не собираетесь дожидаться.

Функция *IoBuildSynchronousFsdRequest* создает *IRP*, завершения которого вы намерены дождаться.

Функция *IoBuildDeviceControlRequest* создает синхронный запрос *IRP\_MJ\_DEVICE\_CONTROL* или *IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL*.

Функция *IoAllocateIrp* создает асинхронный *IRP* любого типа.

### 3.4 Создание синхронных *IRP*

Синхронный *IRP* принадлежит потоку, в контексте которого он был создан. Это определяет следующие его особенности:

- При завершении потока диспетчер ввода-вывода автоматически отменяет все незавершенные синхронные *IRP*, принадлежащие этому потоку.
- Если синхронный *IRP* был создан в произвольном потоке, то он будет отменен в случае завершения этого потока.

- После вызова функции *IoCompleteRequest* диспетчер ввода-вывода отменяет *IRP* и выдает заранее подготовленное драйвером событие.
- Программист драйвера должен обеспечить существование объекта события на момент выдачи его диспетчером ввода-вывода.

Следует также следить за тем, чтобы функции вызывались только на пользовательском уровне прерываний (*PASSIVE\_LEVEL*).

Рассмотрим пример, который приведет к состоянию блокировки.

```
PIRP Irp = IoBuildSynchronousFsdRequest(...);
ExAcquireFastMutex(...);    // Захват мьютекса Здесь IRQL
                             // переключается на APC_LEVEL
NTSTATUS status = IoCallDriver(...);
if (status == STATUS_PENDING)
    KeWaitForSingleObject(...);    // !!!
ExReleaseFastMutex(...);    // Освобождение мьютекса
```

После завершения *IRP* функция *IoCompleteRequest* запланирует *APC* (асинхронный вызов процедуры) для завершения начатого. *APC*, если бы она смогла, выполниться, инициировала бы событие. Однако, выполниться она не может, так как захват мьютекса уже перевел *IRQL* на *APC\_LEVEL*, который не может быть прерван процедурой того же уровня. В итоге, функция не может быть завершена (не сгенерировано событие), а мьютекс не может быть освобожден (так как драйвер не может вернуть *STATUS\_PENDING*).

Для того, чтобы избежать такой блокировки при синхронизации *IRP*, отправляемых другому драйверу, можно поступить по разному.

- Можно использовать обычный мьютекс, вместо быстрого, который оставляет *IRQL* на уровне *PASSIVE\_LEVEL*. При этом могут выполняться *APC* режима ядра.
- Можно использовать функцию *KeEnterCriticalRegion*, которая отключает обычные *APC* режима ядра, а затем использовать функцию *ExAcquireFastMutexUnsafe* для захвата мьютекса. Эта функция ожидает вызова при отключенной доставке обычных *APC* режима ядра.

- Самый простой выход – использовать асинхронный *IRP*, и инициировать событие в функции завершения.

В таблице 3.1 показаны типы синхронных *IRP*.

Таблица 3.1 – типы синхронных *IRP*

Функция	Типы создаваемых IRP
IoBuildSynchronousFsdRequest	IRP_MJ_READ
	IRP_MJ_WRITE
	IRP_MJ_FLUSH_BUFFERS
	IRP_MJ_SHUTDOWN
	IRP_MJ_PNP
	IRP_MJ_POWER
IoBuildDeviceControlRequest	IRP_MJ_DEVICE_CONTROL
	IRP_MJ_INTERNAL_DEVICE_CONTROL

### 3.5 Создание асинхронных *IRP*

Для создания асинхронных *IRP* используются две другие функции:

Функция	Типы создаваемых IRP
IoBuildAsynchronousFsdRequest	IRP_MJ_READ
	IRP_MJ_WRITE
	IRP_MJ_FLUSH_BUFFERS
	IRP_MJ_SHUTDOWN
	IRP_MJ_PNP
	IRP_MJ_POWER
	(только для IRP_MJ_POWER_SEQUENCE)
IoAllocateIrp	Любые (при этом необходимо инициализировать поле MajorFunction в первом элементе стека)

Асинхронные *IRP* не принадлежат породившему их потоку. Диспетчер ввода-вывода не планирует *ACP* и не выполняет их деинициализацию при завершении *IRP*. Особенности асинхронных *IRP*:

- при завершении потока диспетчер ввода-вывода не пытается отменить все асинхронные *IRP*, созданные в этом потоке,

- асинхронные *IRP* могут создаваться как в произвольных, так и в фиксированных потоках,
- так как диспетчер ввода-вывода не следит за завершением *IRP*, программист драйвера должен подготовить функцию завершения, которая вызовет *IoFreeIrp* для освобождения памяти, занятой *IRP*,
- программист драйвера должен позаботиться об отмене *IRP*, когда надобность в их выполнении отпадет,
- так как завершения асинхронных *IRP* дожидаться не надо, их можно создавать и отправлять на более высоких уровнях *IRQL* ( $\leq$  *DISPATCH\_LEVEL*); при этом надо следить за тем, чтобы тот драйвер, которому направляется этот *IRP*, мог обрабатывать его на повышенном уровне *IRQL*; поэтому же возможны создание и отправка асинхронных *IRP* при захвате быстрого мьютекса.

### 3.6 Передача пакета диспетчерской функции

После создания *IRP* (синхронного или асинхронного) можно вызвать функцию *IoGetNextIrpStackLocation* для получения указателя на первый элемент стека. Затем необходимо инициализировать первый элемент.

Если *IRP* был создан функцией *IoAllocateIrp*, необходимо заполнить поле *MajorFunction* (возможно и другие). При создании *IRP* другими функциями необходимость инициализации первого элемента зависит от типа *IRP* (может оказаться, что всю необходимую инициализацию уже выполнил диспетчер ввода-вывода).

После инициализации стека *IRP* посылается драйверу устройства функцией *IoCallDriver*:

```
PDEVICE_OBJECT DeviceObject    // получено извне
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP_MJ_Xxx;
...    // прочая инициализация стека (если необходимо)
NTSTATUS status = IoCallDriver(DeviceObject, Irp);
```

Первый аргумент функции *IoCallDriver* содержит адрес объекта устройства, полученный одним из указанных выше способов из внешнего источника.

Часто *IRP* посылаются нижележащему (в стеке PnP) драйверу. В этом случае *DeviceObject* содержит значение *LowerDeviceObject*, которое сохраняется в расширении устройства после вызова функции *IoAttachDeviceToDeviceStack*.

Диспетчер ввода-вывода указывает на элемент стека *IRP* за одну позицию до фактической. То есть, фактически он указывает на *минус первый* элемент, которого в природе не существует. Поэтому, если необходим первый элемент стека, необходимо запрашивать *следующий* элемент.

### 3.7 Функция *IoCallDriver*

Типичный внешний вид функции *IoCallDriver* примерно таков:

```
NTSTATUS IoCallDriver(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    IoSetNextIrpStackLocation(Irp);
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    stack->device_object = DeviceObject;
    ULONG fcn = stack->MajorFunction;
    PDRIVER_OBJECT driver = DeviceObject->DriverObject;
    return (*driver->MajorFunction[fcn])(DeviceObject, Irp);
}
```

Как видно из примера, функция *IoCallDriver*:

- перемещает указатель стека,
- вызывает соответствующую диспетчерскую функцию драйвера для целевого объекта устройства,
- возвращает код состояния, полученный от диспетчерской функции.

### 3.8 Диспетчерские функции

Рассмотрим типичную диспетчерскую функцию *IRP*:

```
NTSTATUS DispatchXxx(PDEVICE_OBJECT fdo, PIRP Irp)
{
```



```

PIO_STACK_LOCATION stack =
    IoGetCurrentIrpStackLocation(Irp);           //1
PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) Device-> DeviceExtension; //2
...
return STATUS_XXX;                               //3
}

```

Комментарии:

//1 Для определения параметров и дополнительного кода функции обычно приходится обращаться к текущему элементу стека.

//2 Обычно необходим также доступ к расширению устройства, созданного функцией *AddDevice*.

//3 Диспетчерская функция возвращает некоторый код *NTSTATUS* функции *IoCallDriver*, которая передает его далее.

На месте многоточия в диспетчерской функции должен выполняться один из вариантов действий:

- a. немедленно завершить запрос,
- b. передать его драйверу, расположенному ниже в том же стеке,
- c. поставить запрос в очередь для последующей обработки другими функциями драйвера.

#### a. Завершение *IRP*

Любой *IRP* рано или поздно должен быть завершен. В диспетчерской функции *IRP* завершается в следующих случаях:

- если очевидна ошибочность запроса (например, запрос на перемотку принтера или извлечение клавиатуры); в этом случае диспетчерская функция должна отклонить запрос, выдав соответствующий код состояния,
- если запрос требует информацию, которую может предоставить сама диспетчерская функция (например, получение версии драйвера); в этом случае диспетчерская функция предоставляет запрошенную информацию и завершает запрос с успешным кодом.

Завершение *IRP* сводится к заполнению полей *Status* и *Information* в блоке *IRP IoStatus* и последующему вызову функции *IoCompleteRequest*.

Поле *Status* принимает одно из значений, определенных в заголовочном файле *NTSTATUS.H*. Некоторые стандартные значения состояния приведены в таблице 3.2.

Таблица 3.2 – Стандартные значения *NTSTATUS*

Значение	Описание
STATUS_SUCCESS	Нормальное завершение
STATUS_UNSUCCESSFUL	Запрос завершился неудачно, но точная причина не может быть описана
STATUS_NOT_IMPLEMENTED	Функция не была реализована
STATUS_INVALID_HANDLE	Для выполнения операции был передан недействительный дескриптор
STATUS_INVALID_PARAMETER	Неверное значение параметра
STATUS_INVALID_DEVICE_REQUEST	Запрос недействителен для данного устройства
STATUS_EBD_OF_FILE	Достигнут конец файла
STATUS_DELETE_PENDING	Устройство находится в процессе отключения
STATUS_INSUFFICIENT_RESOURCES	Недостаточно системных ресурсов (обычно памяти)

Завершение запроса в диспетчерской функции встречается наиболее часто. Соответствующая функция может выглядеть следующим образом:

```
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS Status,
    ULONG_PTR Information)
{
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = Information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT)
    return status;
}
```

При вызове функции *IoCompleteRequest* аргументом передается величина приращения приоритета, относящаяся ко всем потокам, ожидающим завершения этого запроса.

Перед вызовом функции *IoCompleteRequest* необходимо удалить все функции отмены, которые могли бы быть связаны с *IRP*. Эта функция решает несколько задач:

- вызов функций завершения, установленных разными драйверами,
- снятие блокировки со страниц *MDL* (*Memory Descriptor List*), ассоциированных с *IRP* (они связаны с буферами запросов чтения и записи),
- планирование специального *APC* режима ядра для выполнения итоговой зачистки.

В итоговую зачистку, в частности, входит:

- копирование входных данных обратно в пользовательский буфер,
- копирование итогового состояния *IRP*,
- установка события, которого ожидает создатель *IRP*.

### 3.9 Передача *IRP* вниз по стеку

Возможность простой передачи *IRP* с одного уровня на следующий, низший, обеспечивает создаваемая иерархия объектов устройств.

Создание стека объектов устройств обеспечивается, в частности, функцией *AddDevice*:

```
pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
```

Здесь *fdo* – адрес вашего объекта устройства, а *pdo* – адрес физического объекта устройства на нижнем уровне стека устройств. Функция *IoAttachDeviceToDeviceStack* возвращает адрес объекта устройства, расположенного непосредственно под вашим объектом. Для передачи *IRP*, полученного с высшего уровня, именно этот объект устройства указывается в функции *IoCallDriver*.

При передаче *IRP* на низший уровень, на программиста драйвера возлагаются обязанности по инициализации структуры *IO\_STACK\_LOCATION*. Данные из этой структуры используются следующим драйвером для получения параметров.

Простейший способ инициализации – физическое копирование:

...

```
IoCopyCurrentIrpStackLocationToNext(Irp);
```

```
status = IoCallDriver(pdx->LowerDeviceObject, Irp);
```

...

*IoCopyCurrentIrpStackLocationToNext* копирует все поля *IO\_STACK\_LOCATION* из текущей позиции в следующую. При этом не копируются поля, связанные с функциями завершения ввода-вывода.

### 3.10 Постановка *IRP* в очередь для последующей обработки

Постановка *IRP* в очередь для последующей обработки – третий возможный вариант поведения диспетчерской функции. Приведенный ниже фрагмент кода предполагает, что для работы с очередями *IRP* используется объект *DEVQUEUE*:

```
NTSTATUS DispatchSomething(PDEVICE_OBJECT fdo, PIRP Irp)
{
    ...
    IoMarkIrpPending(Irp);    //a
    StartPacket(&pdx->dqSomething, fdo, Irp, CancelRoutine); //b
    return STATUS_PENDING;    //c
}
```

- a. Эту функцию следует вызывать всегда, когда диспетчерская функция возвращает *STATUS\_PENDING*.
- b. Если устройство в данный момент занято или остановлено из-за события PnP или управления питанием, функция *StartPacket* помещает запрос в очередь; в противном случае устройство помечается как занятое, и вызывается функция *StartIo*.
- c. Возврат *STATUS\_PENDING* говорит о том, что обработка *IRP* еще не завершена.

Следует отметить, что после вызова *StartPacket* не следует обращаться к *IRP*, так как к моменту возврата управления из этой функции *IRP* может оказаться завершенным, а занимаемая им память – освобожденной.

### 3.11 Функция *StartIo*

Функция *StartIo* часто используется для обработки *IRP* в очередях. В приведенном ниже примере собственно обработка *IRP* изображена многоточием.

```
VOID StartIo(PDEVICE_OBJECT device, PIRP Irp)
```

```

{
PIO_STACK_LOCATION stack = IoGetCurrentStackLocation(Irp);
PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) device->DeviceExtension;
...
}

```

Задача функции *StartIo* инициировать обработку *IRP*. Способ обработки полностью зависит от типа устройства. Необходимо иметь в виду, что эта функция получает управление на уровне *DISPATCH\_LEVEL*.

### 3.12 Обработчик прерывания *ISR*

Завершив пересылку данных, устройство может выдать аппаратное прерывание. Обработчик прерывания работает на уровне *IRQL* конкретного устройства (*DIRQL*) и защищается спин-блокировкой.

Спин-блокировки реализуются различным образом для многопроцессорных и однопроцессорных систем. Эти блокировки используются для доступа к общему ресурсу.

Для многопроцессорных систем: код, выполняемый на процессоре А, обращается к общему ресурсу в некоторый момент *t1*. Он устанавливает спин-блокировку и начинает работать с ресурсом. Если в следующий момент времени *t2* код, выполняемый на процессоре Б, также захочет получить доступ к этому ресурсу, он захочет захватить спин-блокировку. Так как спин-блокировка принадлежит процессору А, процессор Б входит в активный цикл, постоянно проверяя состояние ресурса, и ожидая его освобождения. Когда процессор А снимает спин-блокировку, процессор Б обнаруживает, что ресурс освободился, и захватывает его. С этого момента процессор Б имеет неограниченный доступ к ресурсу. После завершения работы с ресурсом, процессор Б освобождает блокировку.

В однопроцессорных системах спин-блокировка реализована существенно проще. Здесь установление спин-блокировки повышает уровень *IRQL* до уровня *DISPATCH\_LEVEL*. В этом случае блокировка обеспечивается тем фактом, что

операции на уровнях *IRQL* выше *PASSIVE\_LEVEL* не могут прерываться другими операциями, работающими на том же или более низком уровне.

Типичный код *ISR* выглядит следующим образом:

```
BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject.  
    PDEVICE_EXTENSION pdx)  
{  
    if <устройство не выдавало прерывания>  
        return FALSE;  
    ...  
}
```

Первый аргумент *ISR* содержит адрес объекта прерывания, созданного функцией *IoConnectInterrupt*.

Второй аргумент указывает на контекст. Чаще всего, это адрес расширения устройства.

Обработчик прерываний в основном отвечает за чтение и запись данных. При этом в большинстве случаев в его обязанности входит планирование отложенного вызова процедуры (*DPC – Deferred Procedure Call*). Помимо основного назначения *DPC* – быстрее освободить механизм прерываний – в *DPC* можно выполнять действия, которые не могут быть выполнены на уровне обработчика (*DIRQL*), например, вызвать *IoCompleteRequest*.

Для этого в обработчике может присутствовать строка типа:

```
IoRequestDpc (pdx->DeviceObject, NULL, pdx);
```

### 3.13 Функция *DPC*

Функция *DpcForIsr*, запрашиваемая обработчиком прерывания, получает управление на уровне *DISPATCH\_LEVEL*. Обычно она должна обработать *IRP*, ставший причиной последнего прерывания. Для этого приходится вызывать функцию *IoCompleteRequest* для завершения *IRP*, а также функцию *StartNextPacket* для исключения следующего *IRP* из очереди устройств и передачи его функции *StartIo*.

Например, так:

```
VOID DpcForIsr (PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk,
```

```

PDEVICE_EXTENSION pdx)
{
...
StartNextPacket(&pdx->dqSomething, fdo);    //a
IoCompleteRequest(Irp, boost);              //b
}

```

- a. *StartNextPacket* удаляет следующий *IRP* из очереди и посылает его *StartIo*,
- b. *IoCompleteRequest* завершает *IRP*, указанный в первом аргументе; второй аргумент задает приращение приоритета для потока, ожидающего этого *IRP*.

### 3.14 Функции завершения

Для получения драйвером вышележащего уровня информации о результатах выполнения конкретного запроса ввода-вывода *IRP* драйвером нижележащего уровня (*CompletionNotification*) можно вызвать функцию *IoSetCompletionRoutine()*:

```

VOID IoSetCompletionRoutine(IN PIRP Irp,
IN PIO_COMPLETION_ROUTINE CompletionRoutine,
IN PVOID Context,
IN BOOLEAN InvokeOnSuccess,
IN BOOLEAN InvokeOnError,
IN BOOLEAN InvokeOnCancel);

```

Здесь *Irp* – указатель на *IRP*, информация о завершении которого необходима, *CompletionRoutine* – указатель на точку входа драйвера, вызываемую при завершении *IRP*, *Context* – определенное драйвером значение, которое нужно передать как третий параметр для точки входа *CompletionRoutine*, *InvokeOnSuccess*, *InvokeOnError*, *InvokeOnCancel* – параметры, которые указывают, должна ли точка входа *CompletionRoutine* быть вызвана при завершении *IRP* с указанным состоянием.

Как видно из прототипа функции, аргументы *InvokeOnXxx* представляют собой логические значения, которые указывают, в какой из трех ситуаций должна вызываться функция завершения.

*InvokeOnSuccess* означает, что функция завершения должна вызываться при завершении *IRP* с кодом состояния, проходящим проверку *NT\_Success*.

*InvokeOnError* означает, что функция завершения должна вызываться при завершении *IRP* с кодом состояния, не 1проходящим проверку *NT\_Success*.

*IwokeOnCancel* означает, что функция должна вызываться в том случае, если перед завершением была вызвана функция *IoCancelIrp*. На самом деле при истинном значении параметра *IwokeOnCancel* проверяется флаг *Cancel* в *IRP*, который и устанавливается функцией *IoCancelIrp*. Если пакет *IRP* завершается с ошибкой, и истинным является параметр *InvokeOnError*, также будет вызвана функция завершения. Если истинным является параметр *InvokeOnSuccess*, функция завершения будет вызвана при завершении пакета *IRP* без ошибки.

Хотя бы один из трех указанных параметров должен быть истинным. Если истинными являются все три параметра, функция завершения будут вызываться при любом завершении пакета *IRP*.

В качестве второго параметра в вызове *IoSetCompletionRoutine()* передается адрес точки входа драйвера, которая должна быть вызвана при завершении указанного в первом параметре пакета *IRP*. Прототип функции – точки входа драйвера:

```
NTSTATUS CompletionRoutine(IN PDEVICE_OBJECT DeviceObject,  
    IN PIRP Irp,  
    IN PVOID Context);
```

Здесь *DeviceObject* – указатель на объект-устройство, которому предназначался завершившийся пакет *IRP*, *IRP* – указатель на завершившийся пакет *IRP*, *Context* – определенное драйвером значение контекста, переданное, когда была вызвана функция *IoSetCompletionRoutine()*.

При вызове *IoSetCompletionRoutine()* указатель на функцию завершения сохраняется в *IRP* в следующем после текущего стеке размещения ввода/вывода,



то есть в стеке размещения ввода/вывода нижележащего драйвера. Из этого следуют два важных вывода:

- если драйвер установит функцию завершения для некоторого *IRP* и завершит этот *IRP*, функция завершения не будет вызвана,
- драйвер низшего уровня (либо монолитный драйвер) не может устанавливать функции завершения, так как, во-первых, это бессмысленно (см. предыдущий вывод), а во-вторых, это ошибка, так как следующего (за текущим) стека размещения ввода/вывода для таких драйверов не существует.

Функция завершения вызывается при том же уровне *IRQL*, при котором нижележащим драйвером была вызвана функция завершения обработки *IRP* – *IoComplete Request()*. Это может быть любой уровень *IRQL* меньший либо равный *IRQL\_DISPATCH\_LEVEL*.

Если драйвер вышележащего уровня создавал новые пакеты *IRP* для передачи драйверу нижележащего уровня, он обязан использовать функцию завершения для этих *IRP*, причем параметры *InvokeOnSuccess*, *InvokeOnError*, *IoCancel* должны быть установлены в *TRUE*.

В этих случаях функция завершения должна освободить созданные драйвером *IRP* с помощью функции *IoFreeIrp()* и завершить первоначальный пакет *IRP*.

Функция завершения может возвращать два возможных значения [3]:

*STATUS\_MORE\_PROCESSING\_REQUIRED* (требуется дополнительная обработка) – процесс завершения немедленно отменяется.

Любое другое значение просто разрешает продолжить процесс завершения. Так как все значения, кроме *STATUS\_MORE\_PROCESSING\_REQUIRED* имеют один и тот же смысл, целесообразно использовать код *STATUS\_SUCCESS*.

### 3.15 Вызов функций завершения

За вызов всех функций завершения, установленных данным драйверов в элементах стека ввода-вывода отвечает *IoCompleteRequest*. Схема вызова этих функций показана на рисунке 3.6.

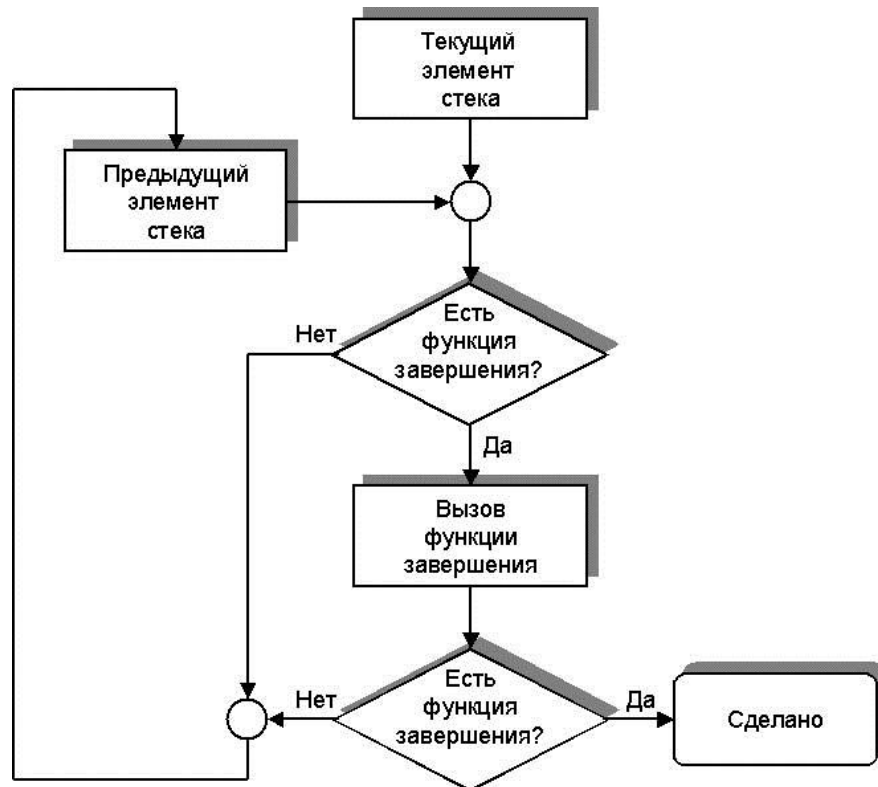


Рисунок 3.6 – Схема вызова функций завершения

На схеме показано, что после вызова функции *IoCompleteRequest* она проверяет текущий элемент стека ввода-вывода на предмет наличия в нем функции завершения, установленной драйвером высшего уровня. Если функция не обнаружена, указатель стека смещается на один уровень вверх, и проверка выполняется снова. Процесс проверки выполняется до тех пор, пока не будет найден элемент стека с установленной функцией завершения, или функция *IoCompleteRequest* не достигнет вершины стека. Затем функция *IoCompleteRequest* выполняет все необходимые действия (в том числе и освобождение памяти, занимаемой *IRP*).

Если функция *IoCompleteRequest* находит позицию стека с указателем на функцию завершения, она вызывает эту функцию и анализирует код возврата. Если код возврата не совпадает с *STATUS\_MORE\_PROCESSING\_REQUIRED*, *IoCompleteRequest* перемещает

указатель стека на один уровень вверх и продолжает работу. Если же код возврата совпадает со значением *STATUS\_MORE\_PROCESSING\_REQUIRED*, функция *IoCompleteRequest* прерывает работу и возвращает управление тому, кто ее вызвал. При этом пакет запроса ввода-вывода *IRP* оказывается в «подвешенном» состоянии. Для вывода пакета из такого состояния драйвер, функция завершения которого прервала процесс «раскрутки» стека, выполнит эту дополнительную обработку *IRP* и опять вызовет функцию *IoCompleteRequest* для продолжения процесса завершения.

Следует отметить, что внутри функции завершения вызов функции *IoGetCurrentIrpStackLocation* получает указатель на элемент стека, который был текущим на момент вызова *IoSetCompletionRoutine*. В функции завершения нельзя пользоваться содержимым элементов стека нижнего уровня. Во избежание такого соблазна функция *IoCompleteRequest* обнуляет большую часть полей следующего элемента перед вызовом функции завершения.

### 3.16 Очереди запросов ввода-вывода

Бывает так, что драйвер получает пакет запроса ввода-вывода *IRP*, который он не может обработать сразу. Диспетчерская функция может отвергнуть этот *IRP* с кодом ошибки, или поместить его в очередь. Далее в драйвере реализуется логика взятия *IRP* из очереди и передачи его функции *StartIo*.

В принципе, организовать очередь *IRP* очень просто. Первый элемент (якорь) списка включается в объект расширения устройства и инициализируется функцией *AddDevice*:

```
typedef struct _DEVICE_EXTENSION {
    LIST_ENTRY IrpQueue;
    BOOLEAN DeviceBusy;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

NTSTATUS AddDevice(...)
{
    ...
    InitializeListHead(&pdx->IrpQueue);
    ...
}
```

```
}
```

После этого пишутся две простые функции для постановки *IRP* в очередь

```
VOID NativeStartPacket(PDEVICE_EXTENSION pdx, PIRP Irp)
{
    if (pdx->DeviceBusy)
        InsertTailList(&pdx->IrpQueue, &Irp->Tail.Overlay.ListEntry);
    else
    {
        pdx->DeviceBusy = TRUE;
        StartIo(pdx->DeviceObject, Irp);
    }
}
```

и извлечения его из очереди

```
VOID NativeStartNextPacket(PDEVICE_EXTENSION pdx, PIRP Irp)
{
    if (IsListEmpty(&pdx->IrpQueue))
        pdx->DeviceBusy = FALSE;
    else
    {
        PLIST_ENTRY foo = RemoveHeadList(&pdx->IrpQueue);
        PIRP Irp = CONTAINING_RECORD(foo, IRP, Tail.Overlay.ListEntry);
        StartIo(pdx->DeviceObject, Irp);
    }
}
```

В коде извлечения *IRP* из очереди используется оригинальный макрос, определенный в `ntdef.h`:

```
#define CONTAINING_RECORD(address, type, field) \
    ((type *)((PCHAR)(address) - (ULONG_PTR)((type *)0)->field)))
```

Этот макрос восстанавливает по известному адресу поля структуры адрес самой структуры.

Рассмотренный метод неудобен тем, что дополнительно необходим механизм приостановки очереди на время некоторых состояний PnP и управления питанием [3]. При этом пакеты запроса ввода-вывода накапливаются в

приостановленной очереди до тех пор, пока кто-нибудь не отменит приостановку очереди, и диспетчер не сможет возобновить отправку *IRP* функции *StartIo*.

### 3.17 Объект *DEVQUEUE*

Для решения проблем, связанных с очередями, в [3] предложен пакет функций управления объектом очереди, который автор назвал *DEVQUEUE* и включил его в структуру *DEVICE\_EXTENSION*.

```
typedef struct _DEVICE_EXTENSION {
    ...
    DEVQUEUE dqReadWrite;
    ...
} DEVICE_EXTENSION, *DEVICE_EXTENSION;
```

На рисунке 3.7 показана типичная обработка *IRP* драйвером, использующим объекты *DEVICE\_EXTENSION* [3].

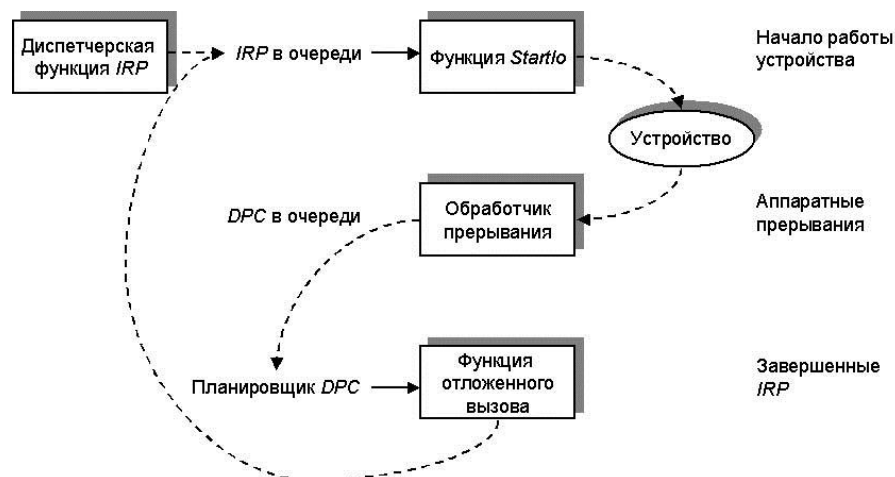


Рисунок 3.7 – Логика обработки *IRP* с участием объекта *DEVQUEUE* и функции *StartIo*

Каждый объект *DEVQUEUE* обладает собственной функцией *StartIo*, которая указывается при инициализации объекта в *AddDevice*.

```
NTSTATUS AddDevice(...)
{
    ...
    PDEVICE_EXTENSION pdx = ...;
    InitializeQueue(&pdx->dqReadWrite, StartIo);
```

```
...  
}
```

Можно использовать общую диспетчерскую функцию для *IRP\_MJ\_READ* и *IRP\_MJ\_WRITE*.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,  
    PUNICODE_STRING RegistryPath)  
{  
    ...  
    DriverObject->MajorFunction[IRP_MJ_READ] = DispatchReadWrite;  
    DriverObject->MajorFunction[IRP_MJ_WRITE] = DispatchReadWrite;  
    ...  
}
```

```
#pragma PAGEDCODE //Следующая функция будет подкачиваемой
```

```
NTSTATUS DispatchReadWrite(PDEVICE_OBJECT fdo, PIRP Irp)  
{  
    PAGED_CODE() //Макрос, убеждающийся в том, что вызывающий  
        //поток запущен на уровне IRQL, который достаточно низок,  
        //чтобы разрешить подкачку  
    PDEVICE_EXTENSION pdx =  
        (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    IoMarkIrpPending(Irp);  
    StartPacket(&pdx->dqReadWrite, fdo, Irp, CancelRoutine);  
    return STATUS_PENDING;  
}
```

```
#pragma LOCKEDCODE //Следующая функция будет заблокированной
```

```
VOID CancelRoutine(PDEVICE_OBJECT fdo, PIRP Irp)  
{  
    PDEVICE_EXTENSION pdx =  
        (PDEVICE_EXTENSION) fdo->DeviceExtension;  
    CancelRequest(&pdx->dqReadWrite, Irp);  
}
```

Следует отметить, что последним аргументом функции *StartPacket* должна быть передана функция отмены.

При завершении *IRP* в функции *DPC* вызывается функция *StartNextPacket*.

```
VOID DpcForIsr(PKDPC junk1, PDEVICE_OBJECT fdo, PIRP junk2,  
    PDEVICE_EXTENSION pdx)  
{  
    ...  
    StartNextPacket(pdx->dqReadWrite, fdo);  
}
```

### 3.18 Отмена запросов ввода-вывода

Достаточно часто возникает необходимость в отмене запроса ввода-вывода. Например, приложение может завершиться после выдачи запроса, но до его выполнения. Появление нового оборудования также может потребовать приостановки запросов, пока менеджер конфигурации не закончит настройку.

Чтобы отменить запрос ввода-вывода в режиме ядра, кто-то должен вызвать функцию *IoCancelIrp*. Система автоматически вызовет *IoCancelIrp* для всех *IRP*, принадлежащих потоку, завершаемому с необработанными запросами.

Приложение пользовательского режима может вызвать *Cancello*, чтобы отменить все незавершенные асинхронные операции, выданные потоком для дескриптора (handle) файла. *IoCancelIrp* могла бы просто завершить указанный *IRP* с кодом завершения *STATUS\_CANCELED*. Однако, функция *IoCancelIrp* не знает, где находятся указатели на *IRP*, и еще она не знает, обрабатывается ли данный *IRP* в данное время, или нет.

Если бы в системе не было многозадачности, и был только один процессор, взаимодействие диспетчера ввода-вывода с функцией *StartIo* было бы достаточно простым:

- При постановке *IRP* в очередь в поле *CancelRoutine* структуры *IRP* заносится адрес функции отмены. При извлечении *IRP* из очереди поле *CancelRoutine* задается равным *NULL*.

- Функция *IoCancelIrp* устанавливает флаг *Cancel* в *Irp*. Затем она проверяет, не равен ли *NULL* указатель *CancelRoutine* в *IRP*. Пока *IRP* находится в очереди, *CancelRoutine* отличен от *NULL*. В этом случае вызов *IoCancelIrp* приводит к вызову установленной функции отмены. Функция отмены удаляет *IRP* из очереди, где он в данный момент находится, и завершает *IRP* с кодом завершения *STATUS\_CANCELED*.
- После извлечения *IRP* из очереди функция *IoCancelIrp* обнаруживает, что указатель *CancelRoutine* равен *NULL*, и вызывает функцию отмены. *IRP* обрабатывается до завершения.

Когда нам приходится работать в реальных условиях многозадачности и многопроцессорности, в которых следствия формально могут опережать причины, может возникать множество ситуаций «гонок» между постановкой в очередь, удалением из очереди и функциями отмены.

Проблемы гонок могут решаться двумя путями. Во-первых, использование собственной логики обработки очередей *IRP*. Во-вторых, использование функций семейства *IoCsqXxx*, предлагаемых фирмой Microsoft. Использование этих функций не предполагает понимания программистом их механизма работы. Фирма предлагает использовать их «как есть».

Рассмотрим примерный вид функции *IoCancelIrp*.

```

BOOLEAN IoCancelIrp(PIRP Irp)
{
    IoAcquireCancelSpinLock(&Irp->CancelIrql);           //1
    &Irp->CancelIrql = TRUE;                               //2
    PDRIVER_CANCEL CancelRoutine = IoSetCancelRoutine(Irp, NULL); //3
    if (CancelRoutine)
    {
        {PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
        (*CancelRoutine)(stack->DeviceObject, Irp);      //4
        return TRUE;
        }
    else
    {
        IoReleaseCancelSpinLock(Irp->CancelIrql);
        return FALSE;                                     //5
    }
}

```



```
}  
}
```

Примечания к коду.

1. Сначала *IoCancelIrps* захватывает глобальную спин-блокировку отмены. Старые драйверы часто использовали эту блокировку, поэтому неизбежно возникали коллизии при попытке ее использования в процессе обычной обработки *IRP*. Новые драйверы используют эту блокировку лишь на короткое время обработки отмены *IRP*.

2. Установка флага *Cancel = TRUE* сообщает всем, что для этого пакета была вызвана функция *IoCancelIrps*.

3. Функция *IoSetCancelRoutine* получает текущий указатель на *CancelRoutine* и задает поле равным *NULL* в течение одной операции.

4. *IoCancelIrps* вызывает функцию отмены, если она присутствует, без предварительной освобождения глобальной спин-блокировки отмены. Поэтому блокировку следует освободить функцией отмены. Следует также обратить внимание на то, что аргумент объекта устройства, передаваемый функции отмены, берется из текущей позиции стека, где он должен был быть оставлен функцией *IoCallDriver*.

5. Если функция отмены отсутствует, функция *IoCancelIrps* сама освобождает глобальную спин-блокировку.

### 3.19 Сценарии обработки *IRP*

Несмотря на все сложности, связанные с обработкой *IRP*, на практике встречаются лишь восемь сценариев, существенно отличающихся друг от друга [3]. Рассмотрим последовательно эти сценарии.

#### *Сценарий 1: передача вниз с функцией завершения*

*IRP*, направленный вашему драйверу, перенаправляется драйверу низшего уровня в стеке PnP, при этом выполняется некоторая заключительная обработка в функции завершения (рисунок 3.8).

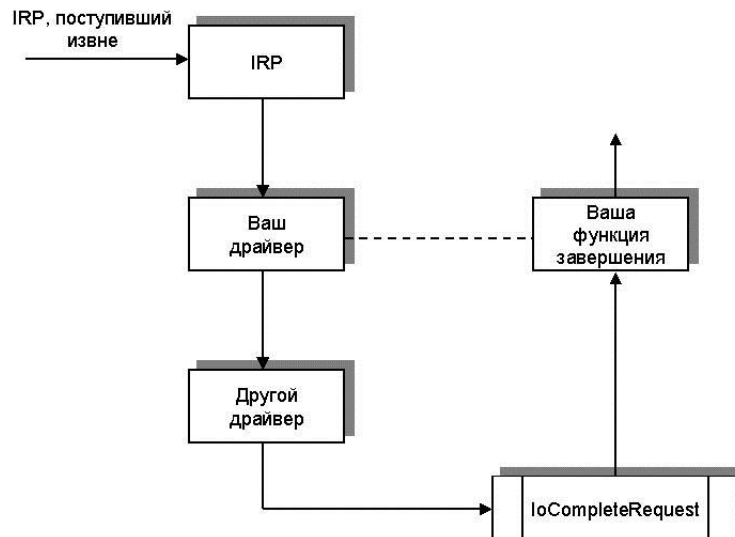


Рисунок 3.8 – Передача вниз с функцией завершения

Для применения этой стратегии необходимы два условия:

- *IRP* может поступить на уровне *DISPATCH\_LEVEL* в контексте произвольного потока (то есть, блокировка во время обработки *IRP* драйверами низшего уровня невозможна),
- если необходимо, заключительная обработка может выполняться на уровне *DISPATCH\_LEVEL* (так как функции завершения могут вызываться на этом уровне).

Код диспетчерской функции и функции завершения может выглядеть примерно так:

```
NTSTATUS DispatchSomething(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx =
        (PDEVICE_EXTENSION) fdo->DeviceExtension;
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status);
    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp,
        (PIO_COMPLETION_ROUTINE) CompletionRoutine,
        pdx, TRUE, TRUE, TRUE);
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
}

NTSTATUS CompletionRoutine(PDEVICE_OBJECT fdo, PIRP Irp,
```

```

PDEVICE_EXTENSION pdx)
{
if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
...    //Необходимая постобработка
IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return STATUS_SUCCESS;
}

```

*Сценарий 2: передача вниз без функции завершения*

*IRP*, направленный вашему драйверу, перенаправляется драйверу низшего уровня в стеке PnP, при этом далее с *IRP* ничего делать не надо (рисунок 3.9).

Для применения этой стратегии необходимы два условия:

- *IRP* получен извне (а не создан вами),
- ваш драйвер не обрабатывает *IRP*, но, возможно, это захочет сделать драйвер низшего уровня.

Этот сценарий часто применяется в фильтрующих драйверах, которые просто передают все *IRP*, не представляющие интереса для данного драйвера.

Код функции, реализующей эту стратегию, может выглядеть примерно так:

```

NTSTATUS ForwardAndForget(PDEVICE_EXTENSION pdx, PIRP Irp)
{
PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);
IoSkipCurrentIrpStackLocation(Irp);
status = IoCallDriver(pdx->LowerDeviceObject Irp);
IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return status;
}

```

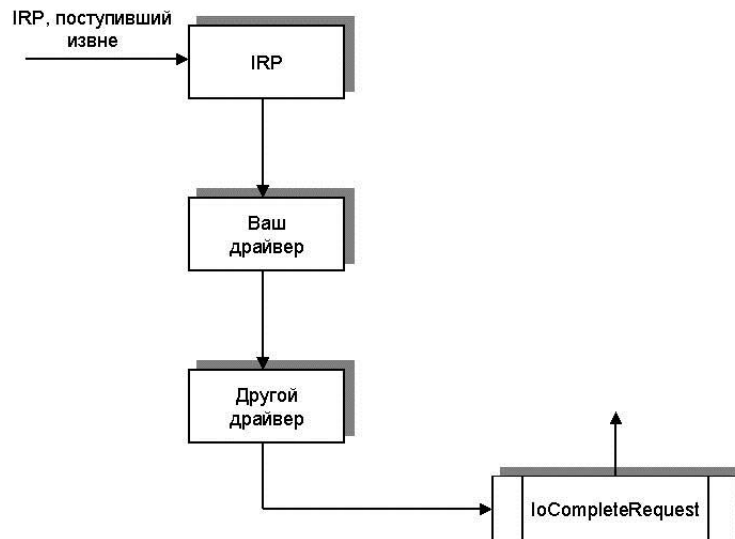


Рисунок 3.9 – Передача вниз без функции завершения

### *Сценарий 3: завершение в диспетчерской функции*

В этом сценарии драйвер немедленно завершает *IRP*, полученный извне (рисунок 3.10). Условия для применения этого сценария:

- *IRP* получен извне (а не создан вами),
- возможна немедленная обработка *IRP*, как это бывает для многих управляющих запросов ввода-вывода (*IOCTL*),
- с пакетом *IRP* что-то очевидно (драйверу) не так, из чего следует немедленный отказ.

Код диспетчерской функции может выглядеть примерно так:

```

NTSTATUS DispatchSomething(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx =
        (PDEVICE_EXTENSION) fdo->DeviceExtension;
    ...          //Необходимая обработка IRP
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = YYY;
    IoSetCompletionRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
  
```

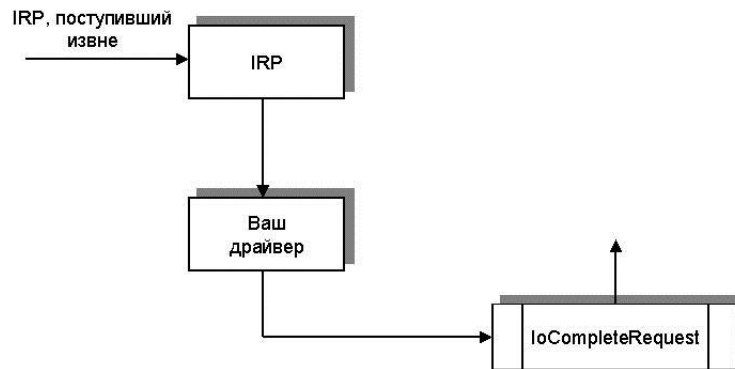


Рисунок 3.10 – Завершение в диспетчерской функции

*Сценарий 4: постановка в очередь для последующей обработки*

В этом сценарии извне получен *IRP*, который ваш драйвер не может обработать немедленно. *IRP* помещается в очередь для последующей обработки в функции *StartIo* (рисунок 3.11). Условия для применения этого сценария:

- *IRP* получен извне (а не создан вами),
- вы не знаете заранее, возможна ли немедленная обработка *IRP*, как это часто бывает для с *IRP*, требующими последовательного доступа к оборудованию.

Типичный способ реализации этого сценария основан на управлении очередью *IRP* при помощи объекта *DEV1QUEUE*. Приводимые ниже фрагменты кода демонстрируют взаимодействие различных частей драйвера устройства с программируемыми прерываниями ввода-вывода.

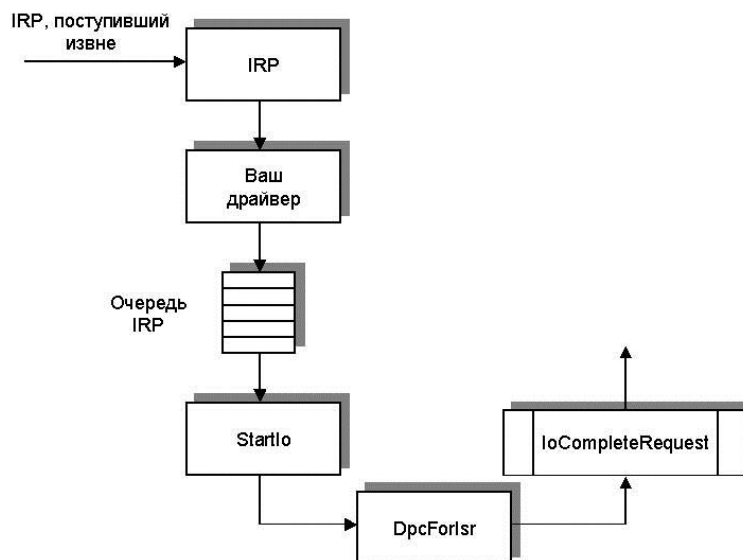


Рисунок 3.11 – Постановка в очередь для последующей обработки

В приведенном ниже коде части, относящиеся непосредственно к обработке *IRP* выделены жирным шрифтом.

```
typedef struct _DEVICE_EXTENSION {
    DEVQUEUE dqReadWrite;
} DEVICE_EXTENSION, *DEVICE_EXTENSION;

NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject,
    PDEVICE_OBJECT pdo)
{
    ...
    InitializeQueue(&pdx->dqReadWrite, StartIo);
    IoInitializeDpcRequest(fdo, (PIO_DPC_ROUTINE) DpcForIsr);
    ...
}

NTSTATUS DispatchReadWrite(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx =
        (PDEVICE_EXTENSION) fdo->DeviceExtension;
    IoMarkIrpPending(Irp);
    StartPacket(&pdx->dqReadWrite, fdo, Irp, CancelRoutine);
    return STATUS_PENDING;
}

VOID CancelRoutine(PDEVICE_OBJECT fdo, PIRP Irp)
{
    PDEVICE_EXTENSION pdx =
        (PDEVICE_EXTENSION) fdo->DeviceExtension;
    CancelRequest(&pdx->dqReadWrite, Irp);
}

VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    ...
}

BOOLEAN OnInterrupt(PKITERRUPT junk, PDEVICE_EXTENSION pdx);
{
    ...
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    Irp->IoStatus.Status = STATUS_XXX;
}
```

```

Irp->IoStatus.Information = YYY;
IoRequestDpc(pdx->DeviceObject, NULL, pdx);
...
}
VOID DpcForIsr(PKDPC junk1, PDEVICE_OBJECT fdo, PIRP junk2,
PDEVICE_EXTENSION pdx)
{
...
PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
StartNextPacket(&pdx->dqReadWrite, fdo);
IoCompleteRequest(Irp, IO_NO_INCREMENT);
}

```

### Сценарий 5: создание асинхронных *IRP*

В этом сценарии драйвер создает асинхронный *IRP*, который пересылается другому драйверу (рисунок 3.12). Условия для применения этого сценария:

- имеется другой драйвер, выполняющий операцию по вашему поручению,
- выполнение ведется либо в контексте произвольного потока (в котором блокировка нежелательна), либо на уровне *DISPATCH\_LEVEL* (на котором блокировка невозможна).

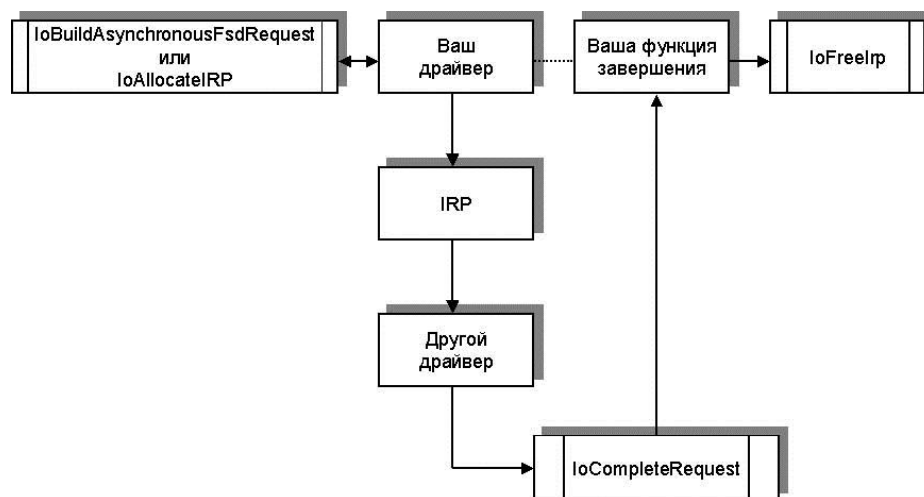


Рисунок 3.12 – Создание асинхронных *IRP*

Ниже приведен примерный код, включаемый в драйвер. Этот код не обязан находиться в диспетчерской функции *IRP*, а целевой объект устройства не обязан быть следующим нижним объектом в стеке.

```

SOMETYPE SomeFunction(PDEVICE_EXTENSION pdx,
PDEVICE_OBJECT DeviceObject)

```

```

{
NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock,
    (PVOID) 42); //A
if (!NT_SUCCESS(status)) //A
    return <состояние>; //A
PIRP Irp;
Irp = IoBuildAsynchronousFsdRequest(IRP_MJ_XXX, DeviceObject,
    ...);

    ИЛИ

Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IPR_MJ_XXX;
<дополнительная инициализация>
IoSetCompletionRoutine[EX]([pdx->DeviceObject,] Irp,
    (PIO_COMPLETION_ROUTINE)CompletionRoutine, pdx,
    TRUE, TRUE, TRUE);
ObReferenceObject(DeviceObject); //B
IoCallDriver(DeviceObject, Irp);
ObDeReferenceObject(DeviceObject); //B
}
NTSTATUS CompletionRoutine(PDEVICEOBJECT junk, PIRP Irp,
    PDEVICETXTENSION PDX)
{
    <зачистка IRP>
    IoFreeIrp(Irp);
    IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42); //A
    return STATUS_MORE_PROCESSING_REQUIRED;
}

```

Вызовы *IoAcquireRemoveLock* и *IoReleaseRemoveLock* (ветвь A) необходимы тогда, когда устройство назначения *IRP* является устройством нижнего уровня в стеке. 42 – просто произвольный маркер.

Вызовы *ObDeReferenceObject* и *ObDeReferenceObject* до и после вызова функции *IoCallDriver* (ветвь B) необходимы тогда, когда функция *IoGetDeviceObjectPointer* использовалась для получения *DeviceObject*, и



функция завершения освобождает полученную ссылку на объект файла или устройства.

Ветви (A) и (B) не используются одновременно – в коде присутствует либо одна ветвь, либо ни одной.

#### *Сценарий 6: создание синхронных IRP*

В этом сценарии драйвер создает синхронный *IRP*, который пересылается другому драйверу (рисунок 3.13). Условия для применения этого сценария:

- имеется другой драйвер, выполняющий операцию по вашему поручению,
- вы должны дождаться завершения операции для продолжения работы.

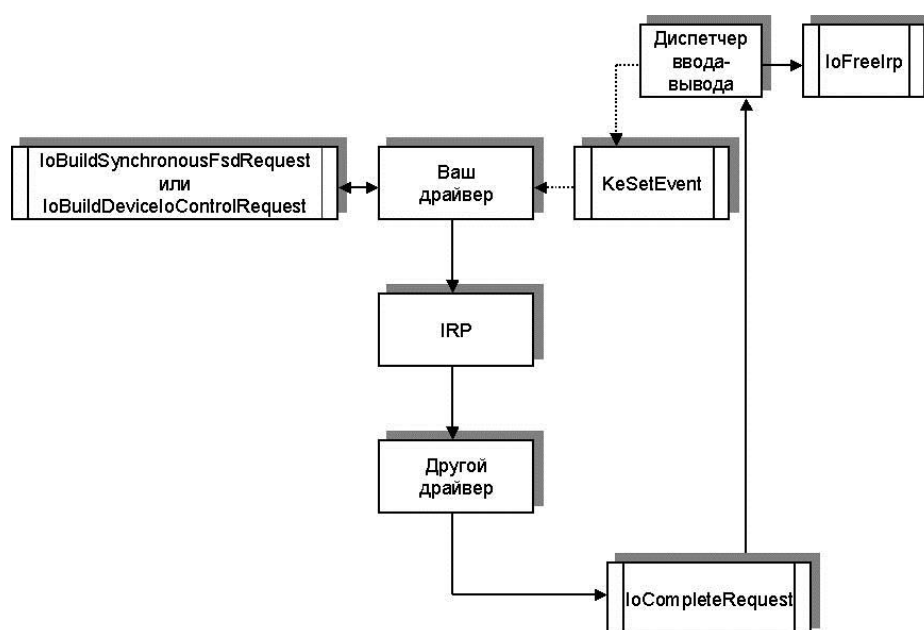


Рисунок 3.13 – Создание синхронных *IRP*

Ниже приведен примерный код, включаемый в драйвер. Этот код не обязан находиться в диспетчерской функции *IRP*, а целевой объект устройства не обязан быть следующим нижним объектом в стеке.

```
SOMETYPE SomeFunction(PDEVICE_EXTENSION pdx,
    PDEVICE_OBJECT DeviceObject)
{
    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock,
        (PVOID) 42); //A
    if (!NT_SUCCESS(status)) //A
        return <состояние>; //A
    PIRP Irp;
    KEVENT event;
```

```

IO_STATUS_BLOCK iosb;
KeInitializeEvent(&event, NotificationEvent, FALSE);
Irp = IoBuildSynchronousFsdRequest(IRP_MJ_XXX, DeviceObject,
... &event, &iosb);

    ИЛИ

Irp = IoBuildDeviceIoControl(IOCTL_XXX, DeviceObject,
... &event, &iosb);
status = IoCallDriver(DeviceObject, Irp);
if (status == STATUS_PENDING)
{
    KeWaitForSingleObject(&event, Executive, KernelMode,
        FALSE, NULL);
    status = iosb.Status;
}
IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42);           //A
...
}

```

Как и в сценарии 5, вызовы *IoAcquireRemoveLock* и *IoReleaseRemoveLock* (комментарии А) необходимы тогда, когда устройство назначения *IRP* является устройством нижнего уровня в стеке. 42 – просто произвольный маркер.

Этот сценарий часто используется для синхронной отправки блоков запросов *USB (URB – USB Request Block)*. Обычно это происходит в контексте диспетчерской функции *IRP*, независимо устанавливающей блокировку удаления. Зачистка за такими *IRP* не выполняется. Диспетчер ввода-вывода выполняет ее автоматически.

#### *Сценарий 7: синхронная передача вниз*

В этом сценарии извне получен *IRP*. Ваш драйвер синхронно передает его вниз по стеку и продолжает работу (рисунок 3.14). Условия для применения этого сценария:

- *IRP* получен извне (а не создан вами),
- выполнение ведется на уровне *PASSIVE\_LEVEL* в фиксированном потоке,

- ваша заключительная обработка этого *IRP* должна выполняться на уровне *PASSIVE\_LEVEL*.

Типичным примером ситуации, в которой необходимо использовать этот сценарий – обработка запросов *PnP* подвида *IRP\_MN\_START\_DEVICE*.

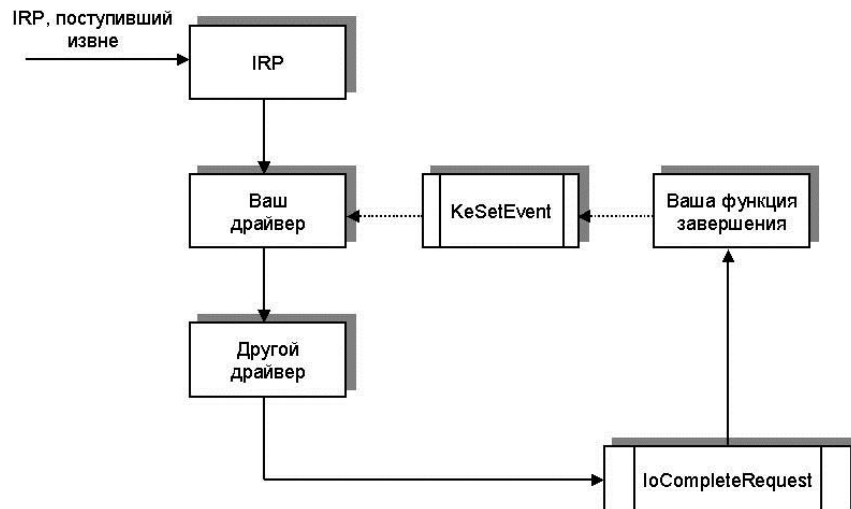


Рисунок 3.14 – Синхронная передачи *IRP* вниз по стеку

Две вспомогательные функции, показанные ниже, сильно упростят выполнение синхронной передачи.

Тот, кто вызывает эту функцию, должен вызвать *IoCompleteRequest* для *IRP*, захватить и освободить блокировку удаления. Логика блокировки и удаления не следует размещать в *ForwardAndWait*, потому что это может вызвать конфликт с вызывающей стороной.

```

NTSTATUS ForwardAndWait(PDEVICE_EXTENSION pdx, PIRP Irp)
{
    KEVENT event;
    KeInitialize(&event, NotificationRoutine, FALSE);
    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
        ForwardAndWaitCompletionRoutine, &event, TRUE, TRUE, TRUE);
    NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    if (status == STATUS_PENDING)
    {
        KeWaitForSingleObject(&event, Executive, KernelMode,
            FALSE, NULL);
        status = Irp->IoStatus.Status;
    }
}
  
```

```

    }
    return status;
}

NTSTATUS ForwardAndWaitCompletionRoutine(PDEVICE_OBJECT fdo,
    PIRP Irp, PKEVENT pev)
{
    if (Irp->PendingReturned)
        KeSetEvent(pev, IO_NO_INCREMENT, FALSE);
    return STATUS_MORE_PROCESSING_REQUIRED;
}

```

### Сценарий 8: синхронная обработка асинхронных *IRP*

В этом сценарии драйвер создает асинхронный *IRP*, пересылает его другому драйверу и ожидает завершения *IRP* (рисунок 3.15). Условия для применения этого сценария:

- имеется другой драйвер, выполняющий операцию по вашему поручению,
- вы должны дождаться завершения операции для продолжения работы,
- выполнение осуществляется на уровне *DPC\_LEVEL* в контексте фиксированного потока.

Типичный пример ситуации, в которой целесообразно использовать этот сценарий – выполнение синхронной операции после захвата быстрого мьютекса.

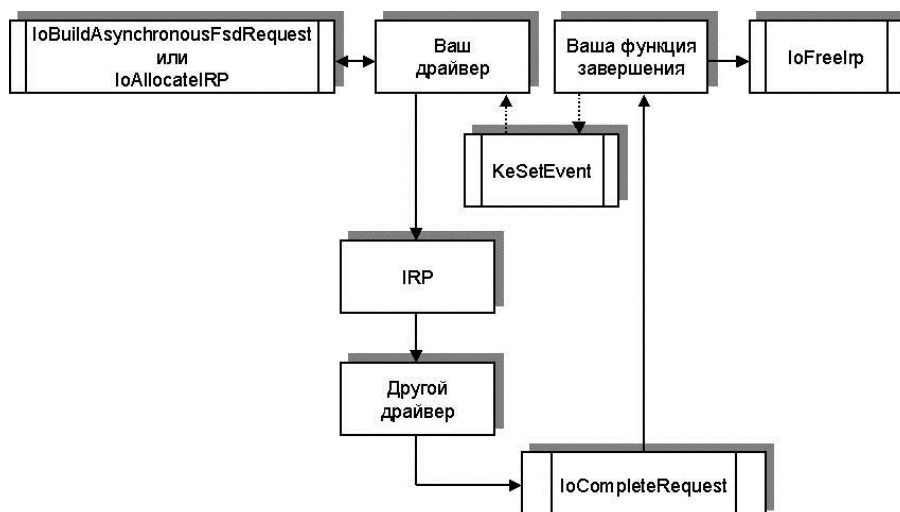


Рисунок 3.15 – Синхронная обработка асинхронных *IRP*

В приведенном коде сочетаются уже встречавшиеся ранее элементы.

```

SOMETYPE SomeFunction(PDEVICE_EXTENSION pdx,
    PDEVICE_OBJECT DeviceObject)

```

```

{
NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock,
    (PVOID) 42);                                //A
if (!NT_SUCCESS(status))                        //A
    return <состояние>;                          //A
PIRP Irp;
Irp = IoBuildAsynchronousFsdRequest(IRP_MJ_XXX, DeviceObject,
    ...);

    ИЛИ

Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
Stack->MajorFunction = IPR_MJ_XXX;
<дополнительная инициализация>
KEVENT event;
KeInitializeEvent(&event, NotificationEvent, FALSE);
IoCopyCurrentIrpStackLocationToNext(Irp);
IoSetCompletionRoutine[EX]([pdx->DeviceObject,] Irp,
    (PIO_COMPLETION_ROUTINE)CompletionRoutine, &event,
    TRUE, TRUE, TRUE);
status = IoCallDriver(DeviceObject, Irp);
if (status == STATUS_PENDING)
    KeWaitForSingleObject(&event, Executive, KernelMode,
        FALSE, NULL);
IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42);    //A
}

NTSTATUS CompletionRoutine(PDEVICE_OBJECT junk, PIRP Irp,
PKEVENT pev)
if (Irp->PendingReturned)
    KeSetEvent(pev, EVENT_INCREMENT, FALSE);
<зачистка IRP - см. выше>
IoFreeIrp(Irp);
return STATUS_MORE_PROCESSING_REQUIRED;
}

```

Фрагменты, отличающиеся от сценария 5, выделены жирным шрифтом. Здесь, как и в предыдущих сценариях, вызовы *IoAcquireRemoveLock* и

*IoReleaseRemoveLock* (комментарии А) необходимы тогда, когда устройство назначения *IRP* является устройством нижнего уровня в стеке. 42 – просто произвольный маркер.

Здесь также необходимо помнить о зачистке, так как диспетчер ввода-вывода не выполняет автоматическую зачистку для асинхронных *IRP*.

### 3.20 Вопросы для самопроверки

1. Что такое пакет запроса ввода-вывода *IRP*?
2. Какова структура пакета запроса ввода-вывода *IRP*?
3. Для чего нужно поле *MdlAddress* в пакете запроса ввода-вывода *IRP*?
4. Для чего нужно поле *Flags* в пакете запроса ввода-вывода *IRP*?
5. Для чего нужно поле *AssociatedIrp* в пакете запроса ввода-вывода *IRP*?
6. Для чего нужно поле *IoStatus* в пакете запроса ввода-вывода *IRP*?
7. Для чего нужно поле *RequestorMode* в пакете запроса ввода-вывода *IRP*?
8. Для чего нужно поле *Cancel* в пакете запроса ввода-вывода *IRP*?
9. Для чего нужно поле *PendingReturned* в пакете запроса ввода-вывода *IRP*?
10. Для чего нужно поле *CansellIrp* в пакете запроса ввода-вывода *IRP*?
11. Для чего нужно поле *CancelRoutine* в пакете запроса ввода-вывода *IRP*?
12. Для чего нужно поле *UserBuffer* в пакете запроса ввода-вывода *IRP*?
13. Для чего нужно поле *Tail* в пакете запроса ввода-вывода *IRP*?
14. В каком случае создается MDL?
15. Что описывает MDL?
16. Что такое стек ввода-вывода?
17. Что собой представляет структура *IO\_STACK\_LOCATION*?
18. Для чего нужно поле *MajorFunction* в структуре *IO\_STACK\_LOCATION*?
19. Для чего нужно поле *MinorFunction* в структуре *IO\_STACK\_LOCATION*?
20. Для чего нужно поле *Flags* в структуре *IO\_STACK\_LOCATION*?
21. Для чего нужно поле *Control* в структуре *IO\_STACK\_LOCATION*?
22. Для чего нужно поле *Parameters* в структуре *IO\_STACK\_LOCATION*?
23. Для чего нужно поле *DeviceObject* в структуре *IO\_STACK\_LOCATION*?
24. Для чего нужно поле *FileObject* в структуре *IO\_STACK\_LOCATION*?
25. Для чего нужно поле *CompletionRoutine* в структуре *IO\_STACK\_LOCATION*?
26. Для чего нужно поле *Context* в структуре *IO\_STACK\_LOCATION*?
27. Какова типичная модель обработки пакета запроса ввода-вывода *IRP*?
28. С чего начинается жизненный цикл *IRP*?
29. Чем завершается жизненный цикл *IRP*?

30. Какая функция создает IRP?
31. Чем отличаются синхронные и асинхронные IRP?
32. Кому принадлежит синхронный IRP?
33. Каковы особенности синхронного IRP?
34. Какие проблемы могут возникать из-за неправильного уровня запроса прерываний IRQL?
35. Когда планируется асинхронный вызов процедуры?
36. Кем планируется асинхронный вызов процедуры?
37. Зачем планируется асинхронный вызов процедуры?
38. В каком случае может возникнуть блокировка при обработке IRP?
39. Как избежать возникновения блокировки при обработке ШКЗ?
40. Какие типы синхронных IRP существуют?
41. Какими функциями могут быть созданы синхронные IRP?
42. Какими функциями могут быть созданы асинхронные IRP?
43. Каковы особенности асинхронных IRP?
44. Какие типы асинхронных IRP существуют?
45. На каких уровнях IRQL могут создаваться асинхронные IRP?
46. На каких уровнях IRQL могут создаваться синхронные IRP?
47. Для чего нужна функция IoFreeIrp?
48. Кто должен вызывать функцию IoFreeIrp?
49. Как пакет запроса ввода-вывода IRP передается диспетчерской функции?
50. Какая функция дает указатель на первый элемент стека ввода-вывода?
51. Почему для получения указателя на первый элемент стека ввода-вывода следует вызывать функцию IoGetNextIrpStackLocation?
52. В каком случае необходимо заполнение поля MajorFunction IRP?
53. Что происходит после инициализации стека ввода-вывода?
54. Какая функция посылает IRP драйверу?
55. Что содержит первый аргумент функции IoCallDriver?
56. Для чего используется функция IoAttachDeviceToDeviceStack?
57. Каков типичный вид функции IoCallDriver?
58. Что делает функция IoCallDriver?
59. Какова типичная диспетчерская функция?
60. Как определяются параметры и дополнительный код функции?
61. Как получается доступ к расширению объекта устройства?
62. Кому диспетчерская функция возвращает код NTSTATUS?
63. Какие еще действия должна выполнить диспетчерская функция?
64. В каких случаях IRP завершается в диспетчерской функции?
65. К чему сводится завершение IRP?

66. Чем заполняется поле Status в блоке IRP IoStatus?
67. Каковы стандартные значения поля Status?
68. Как выглядит функция завершения IRP?
69. Что делает функция CompleteRequest?
70. Для чего в функции IoCompleteRequest передается величина приращения приоритета?
71. К чему относится величина приращения приоритета, которая передается в функции IoCompleteRequest?
72. Что необходимо сделать перед вызовом функции IoCompleteRequest?
73. Какие задачи решает функция IoCompleteRequest?
74. Что такое итоговая зачистка?
75. Как IRP передается вниз по стеку ввода-вывода?
76. Какая функция обеспечивает создание стека устройств?
77. Какие обязанности возлагаются на программиста драйвера при передаче IRP на низший уровень стека?
78. Какие поля IO\_STACK\_LOCATION обычно копируются при передаче IRP на низший уровень?
79. Как IRP ставится в очередь для последующей обработки?
80. Что представляет собой объект DEVQUEUE?
81. Для чего нужен объект DEVQUEUE?
82. Какую функцию следует всегда вызывать, если диспетчерская функция возвращает STATUS\_PENDING?
83. Какая функция помещает IRP в очередь?
84. Какая функция вызывается, если устройство не занято?
85. О чем говорит возврат STATUS\_PENDING?
86. Для чего обычно используется функция StartIo?
87. Какова главная задача функции StartIo?
88. На каком уровне IRQL получает управление функция StartIo?
89. Для чего нужен обработчик прерывания ISR?
90. На каком уровне IRQL работает обработчик прерывания?
91. Что такое спин-блокировка?
92. Как реализуется спин-блокировка для многопроцессорных систем?
93. Как реализуется спин-блокировка для однопроцессорных систем?
94. Как выглядит типичный код ISR?
95. Что содержит первый аргумент ISR?
96. Что содержит второй аргумент ISR?
97. Для чего ISR планирует отложенный вызов процедуры DPC?
98. На каком уровне IRQL получает управление функция DpcForIsr?



99. Что должна сделать функция DpcForIsr?
100. Что такое функция завершения?
101. Как драйвер вышележащего уровня получает информацию о результатах выполнения конкретного запроса ввода-вывода IRP драйвером нижележащего уровня?
102. Что делает функция IoSetCompletionRoutine?
103. Каковы аргументы функции IoSetCompletionRoutine?
104. Для чего необходим первый аргумент функции IoSetCompletionRoutine?
105. Для чего необходим второй аргумент функции IoSetCompletionRoutine?
106. Для чего необходим третий аргумент функции IoSetCompletionRoutine?
107. Для чего необходим четвертый аргумент функции IoSetCompletionRoutine?
108. Для чего необходим пятый аргумент функции IoSetCompletionRoutine?
109. Для чего необходим шестой аргумент функции IoSetCompletionRoutine?
110. Что означает значение аргумента InvokeOnSuccess?
111. Что означает значение аргумента InvokeOnError?
112. Что означает значение аргумента InvokeOnCancel?
113. Каковы требования к трем последним аргументам функции IoSetCompletionRoutine?
114. В каком IRP сохраняется указатель на функцию завершения при вызове функции IoSetCompletionRoutine?
115. Что будет, если драйвер устновит функцию завершения для конкретного IRP и завершит его?
116. На каком уровне IRQL вызывается функция завершения?
117. Что должна сделать функция завершения, если драйвер создавал новые IRP для передачи нижележащему драйверу?
118. Какие значения может возвращать функция завершения?
119. Как вызываются функции завершения, установленные драйвером в элементах стека ввода-вывода?
120. Какая функция отвечает за вызов функций завершения?
121. Когда завершается цикл вызова функций завершения драйвера?
122. В каком случае функция IoCompleteRequest перемещает указатель стека ввода-вывода на один уровень вверх и продолжает работу?
123. В каком случае функция IoCompleteRequest прекращает работу и возвращает управление?
124. Что предусмотрено для вывода IRP из «подвешенного» состояния в случае прекращения работы функции IoCompleteRequest?
125. Что делает функция IoCompleteRequest перед вызовом функции завершения, и зачем?

126. Что делать, если драйвер не может сразу обработать полученный IRP?
127. Как организуются очереди IRP?
128. Куда записывается первый элемент списка очереди IRP?
129. Какая функция используется для постановки IRP очередь?
130. Какая функция используется для извлечения IRP из очереди?
131. Для чего используется макрос CONTAINING\_RECORD?
132. Для чего нужен объект DEVQUEUE?
133. Какова логика обработки IRP с участием объекта DEVQUEUE?
134. Как можно определить общую диспетчерскую функцию для запросов чтения и записи?
135. Для чего нужен макрос PAGED\_CODE?
136. Для чего используется функция IoMarkIrpPending?
137. Для чего используется функция StartPacket?
138. Для чего используется предложение #pragma PAGEDCODE?
139. Для чего нужен макрос LOCKED\_CODE?
140. Для чего используется предложение #pragma LOCKEDCODE?
141. Какой последний аргумент задается в функции StartPacket?
142. Какая функция вызывается при завершении IRP в DPC?
143. В каком случае вызывается функция StartNextPacket?
144. Когда возникает необходимость в отмене запроса ввода-вывода?
145. Как отменить запрос ввода-вывода в режиме ядра?
146. Как отменить запрос ввода-вывода в пользовательском режиме?
147. Как можно решить проблемы «гонок» при отмене IRP?
148. Каковы основные сценарии обработки IRP?
149. При каких условиях IRP передается вниз с функцией завершения?
150. При каких условиях IRP передается вниз без функции завершения?
151. При каких условиях IRP завершается в диспетчерской функции?
152. При каких условиях IRP ставится в очередь для последующей обработки?
153. При каких условиях создаются асинхронные IRP?
154. При каких условиях IRP создаются синхронные IRP?
155. При каких условиях выполняется синхронная передача IRP вниз?
156. При каких условиях выполняется синхронная асинхронных IRP?

*Список использованных источников*

1. Роцин А.В. Организация ввода-вывода. Основы написания драйверов уровня ядра для операционной системы Windows NT5: учебное пособие, часть 3. – М.: МГУПИ, 2009. – 91 с.
2. Харт Дж. М. Системное программирование в среде Windows, 3-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 592 с.
3. Они У. Использование Microsoft Windows Driver Model. 2-е издание. – М.: Питер, 2007. – 764 с.
4. Орвик П. Windows Driver Foundation: разработка драйверов. Пер. с англ. / П.Орвик, Г.Смит. – М.: Издательство «Русская редакция». – СПб.: «БХВ-Петербург», 2008. – 880 с.: ил.

Подписано к печати 21.07.2011 г. Формат 60х84. 1/16  
Объем 5,25 п.л. Тираж 100 экз. Заказ № 120

***Московский государственный университет  
приборостроения и информатики  
107996, Москва, ул. Стромынка, 20***