Глава 7: Пакет запроса ввода/вывода

После того, как типичный драйвер завершает свою инициализацию в DriverEntry, его основная задача — обрабатывать запросы. Эти запросы упакованы в виде полудокументированной структуры пакета запроса ввода-вывода (IRP).

В этой главе мы более подробно рассмотрим пакеты IRP и то, как драйвер обрабатывает стандартные типы пакетов IRP.

В этой главе:

• Введение в IRP.

• Узлы устройств.

• IRP и расположение стека ввода-вывода.

• Процедуры отправки.

• Доступ к пользовательским буферам.

• Собираем все вместе: Нулевой драйвер.

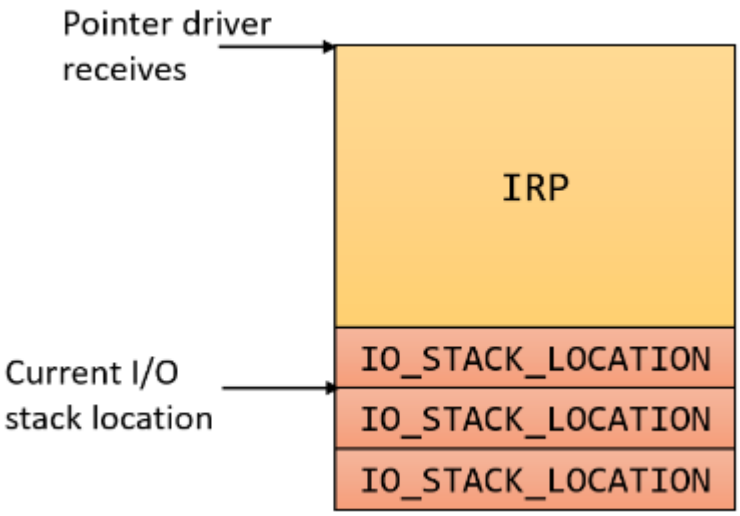
Введение в IRP

IRP - это структура, которая выделяется из невыгружаемого пула, как правило, одним из «менеджеров» (диспетчер ввода-вывода, диспетчер Plug & Play, диспетчер питания), но также может быть выделена драйвером, возможно, для передачи запроса другому драйверу.

Как-бы IRP не выделялся, сервис который выделил эту структуру отвечает за его освобождение.

Когда драйвер получает IRP, он получает указатель на стек ввода-вывода в самой структуре IRP. Зная, что по этому указателю следует набор местоположений стека ввода-вывода, одно из которых предназначено для использования драйвером. Чтобы получить правильное расположение стека ввода-вывода, драйвер вызывает IoGetCurrentIrpStackLocation.

На рисунке показано концептуальное представление пакета IRP и связанного с ним стека ввода-вывода.



Параметры запроса как-то «разбиты» между основной структурой IRP и текущей IO_STACK_LCATION.

Узлы устройств

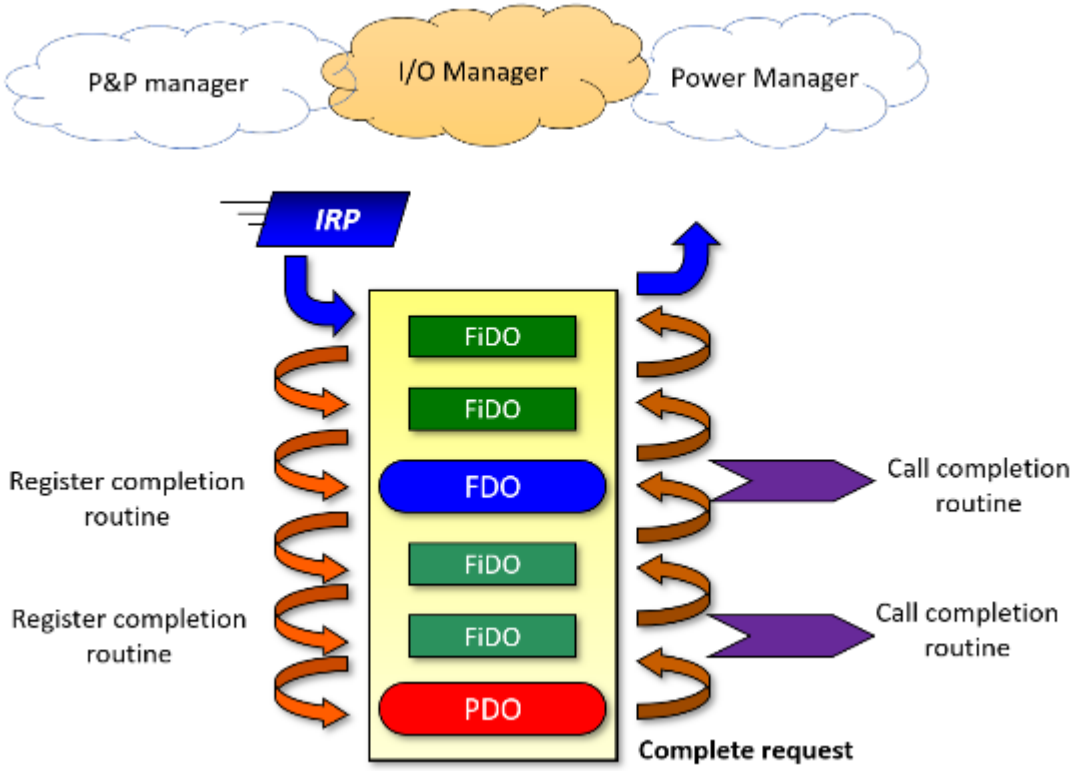
Система ввода-вывода в Windows ориентирована на устройства, а не на драйверы. Это имеет несколько последствий:

• Объектам устройств можно давать имена и открывать дескрипторы объектов устройств.

Функция CreateFile принимает символическую ссылку, ведущую к имени объекта устройства. CreateFile не может принять имя драйвера в качестве аргумента.

• Windows поддерживает наслоение устройств - одно устройство может быть расположено поверх другого. Это означает что любой запрос, адресованный нижнему устройству, сначала достигнет самого верхнего уровня. Это наслоение обычно используется для аппаратных устройств, но работает с любыми типами устройств.

На рисунке ниже показан пример нескольких уровней устройств, «уложенных» одно поверх другого. Этот набор устройств известен как стек устройств, иногда называемый узлом устройства. На рисунке есть шесть слоев или шесть устройств. Каждое из этих устройств на самом деле представляет собой структуру DEVICE_OBJECT, созданную вызовом стандартной функции IoCreateDevice.



Различные объекты устройства, которые составляют уровни узла устройства (devnode), именуются в соответствии с их ролью в devnode. Эти роли актуальны для аппаратного узла устройства.

Вот краткое изложение обозначений, представленных на рисунке выше:

• PDO (Physical Device Object) - Несмотря на название, в нем нет ничего «физического». Это объект устройства создается драйвером шины - драйвером, который отвечает за конкретную шину (например, PCI, USB и т. д.). Этот объект устройства представляет тот факт, что в этом слоте есть какое-то устройство.

• FDO (функциональный объект устройства) - этот объект устройства создается «настоящим» драйвером, это драйвер, который обычно предоставляется поставщиком оборудования, который разбирается в деталях устройства.

• FiDO (объект фильтра) - это дополнительные устройства фильтрации, созданные драйверами фильтров.

Менеджер Plug & Play (P&P) в этом случае отвечает за загрузку соответствующих драйверов, начиная снизу. В качестве примера предположим, что devnode на рисунке выше представляет собой набор драйверов, которые управляют сетевой картой PCI.

Последовательность событий, приведших к созданию этого devnode можно резюмировать следующим образом:

1. Драйвер шины PCI (pci.sys) распознает факт наличия чего-либо в этом конкретном слоте.

Он создает PDO (IoCreateDevice), чтобы указать этот факт. Драйвер шины не знает,

это сетевая карта, видеокарта или что-то еще, он только знает, что там что-то есть и

может извлекать основную информацию из своего контроллера, такую как идентификатор поставщика и идентификатор устройства.

2. Драйвер шины PCI уведомляет диспетчер P&P об изменениях на своей шине.

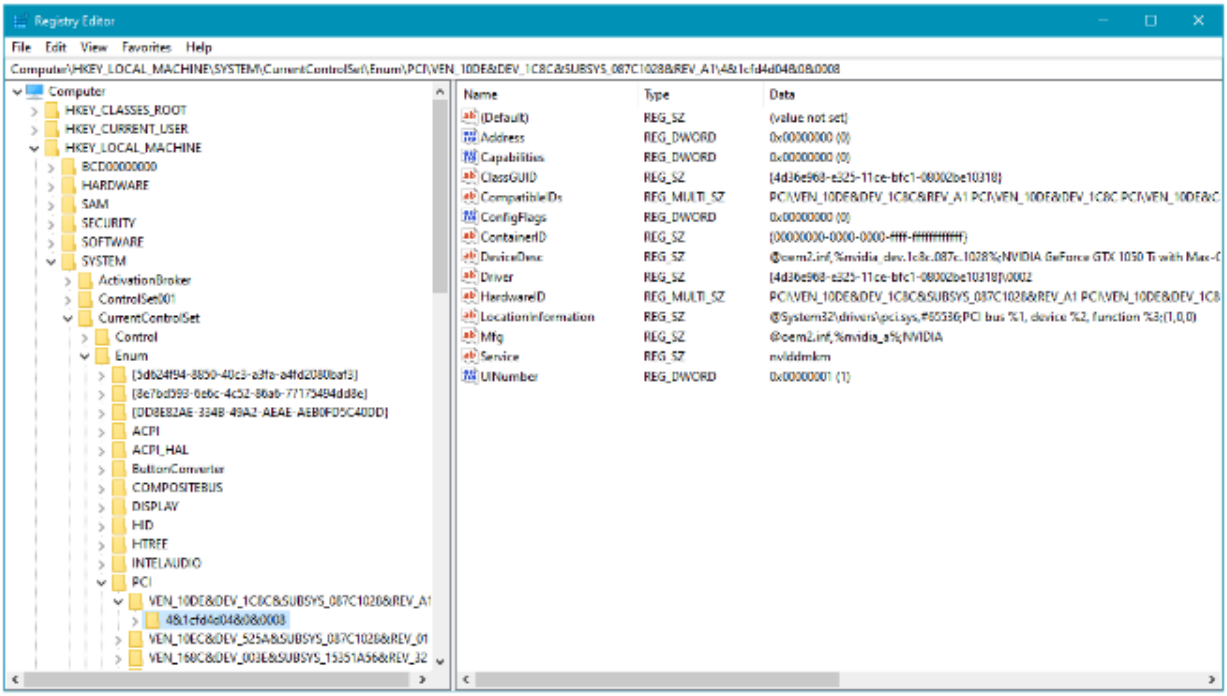
3. Менеджер P&P запрашивает список PDO, управляемых драйвером шины. Он получает обратно список PDO, в которые включен этот новый PDO.

4. Теперь задача менеджера P&P - найти и загрузить соответствующий драйвер для нового PDO. Он выдает запрос к драйверу шины, чтобы запросить полный идентификатор устройства.

5. Имея этот идентификатор оборудования на руках, менеджер P&P просматривает реестр в HKLM\System\CurrentControlSet\Enum\PCI\(Идентификатор оборудования). Если драйвер был загружен раньше, он будет зарегистрирован там, и менеджер P&P его загрузит.

На рисунке ниже показан пример идентификатора оборудования в реестре (драйвер дисплея NVIDIA).

6. Драйвер загружает и создает FDO (еще один вызов IoCreateDevice), но добавляет дополнительный вызов IoAttachDeviceToDeviceStack, таким образом присоединяясь к предыдущему уровню (обычно PDO).



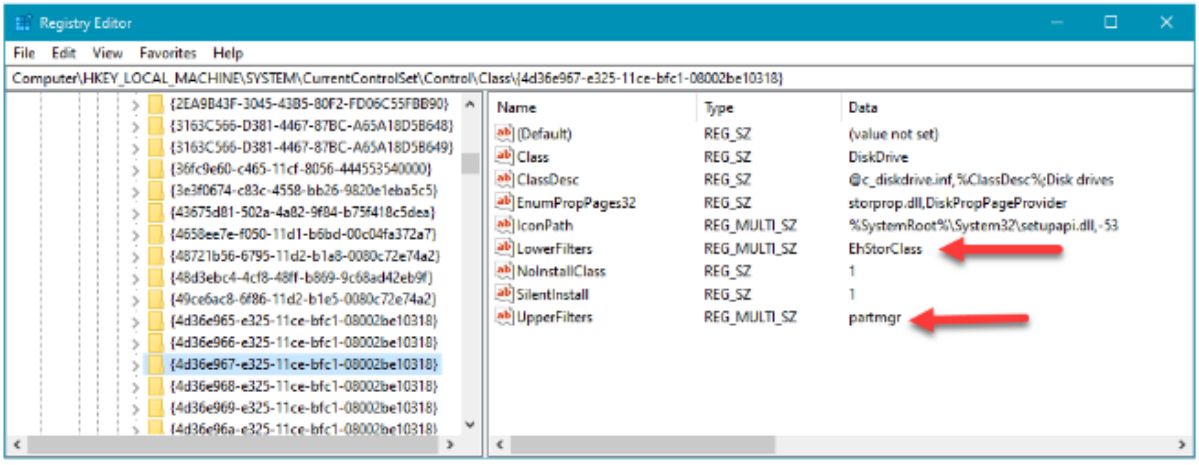
Также загружаются объекты фильтрующих устройств, если они правильно зарегистрированы в реестре. Каждый загруженный драйвер фильтра создает свой собственный объект устройства и прикрепляет его поверх предыдущего слоя. Верхние фильтры работают так же, но загружаются после FDO. Все это означает, что с действующими узлами разработки P&P имеется как минимум два слоя - PDO и FDO, но может быть и больше, если задействованы фильтры.

Разработка аппаратных драйверов будет обсуждаться в главе 11.

Поиск нижних фильтров осуществляется в двух местах: ключ аппаратного идентификатора, показанный на рисунке ниже, и соответствующий класс на основе значения ClassGuid, указанного в HKLMSystemCurrentControlSet\Control\Classes.

Имя фильтра, это значение - LowerFilters. Может-быть указанно несколько фильтров.

Аналогичным образом ищутся верхние фильтры, но имя фильтра будет UpperFilters.



Потоки IRP

IRP создается одним менеджеров исполнения - для большинства наших драйверов это I/O Manager.

Менеджер инициализирует только основную структуру IRP и расположение первого стека ввода-вывода. Затем он передает указатель IRP на самый верхний уровень.

Драйвер получает IRP в соответствующей программе отправки. Например, если это IRP для чтения, тогда драйвер будет вызываться в своем индексе IRP_MJ_READ в массиве MajorFunction из объекта драйвера.

На этом этапе у драйвера есть несколько вариантов при работе с IRP:

1. Передача запроса вниз - если устройство драйвера не является последним устройством в узле разработки, драйвер может передать запрос, если она не интересна драйверу. Обычно это выполняется драйвером фильтра который получает запрос, который его не интересует, и чтобы не повредить функциональность устройства (поскольку запрос на самом деле предназначен для устройства нижнего уровня), драйвер может передать его вниз.

Это нужно сделать двумя вызовами:

• Вызов IoSkipCurrentIrpStackLocation, чтобы убедиться, что следующее устройство в очереди увидит ту-же информацию, что и этому устройству - оно должно видеть то же расположение стека ввода-вывода.

• Вызов IoCallDriver, передавая объект нижнего устройства (который драйвер получил во время вызова IoAttachDeviceToDeviceStack) и IRP.

2. Обработка IRP - драйвер, получающий IRP, может просто обработать IRP, не передовая его вниз, в конечном итоге вызвав IoCompleteRequest. Любые нижние устройства никогда не увидят запрос.

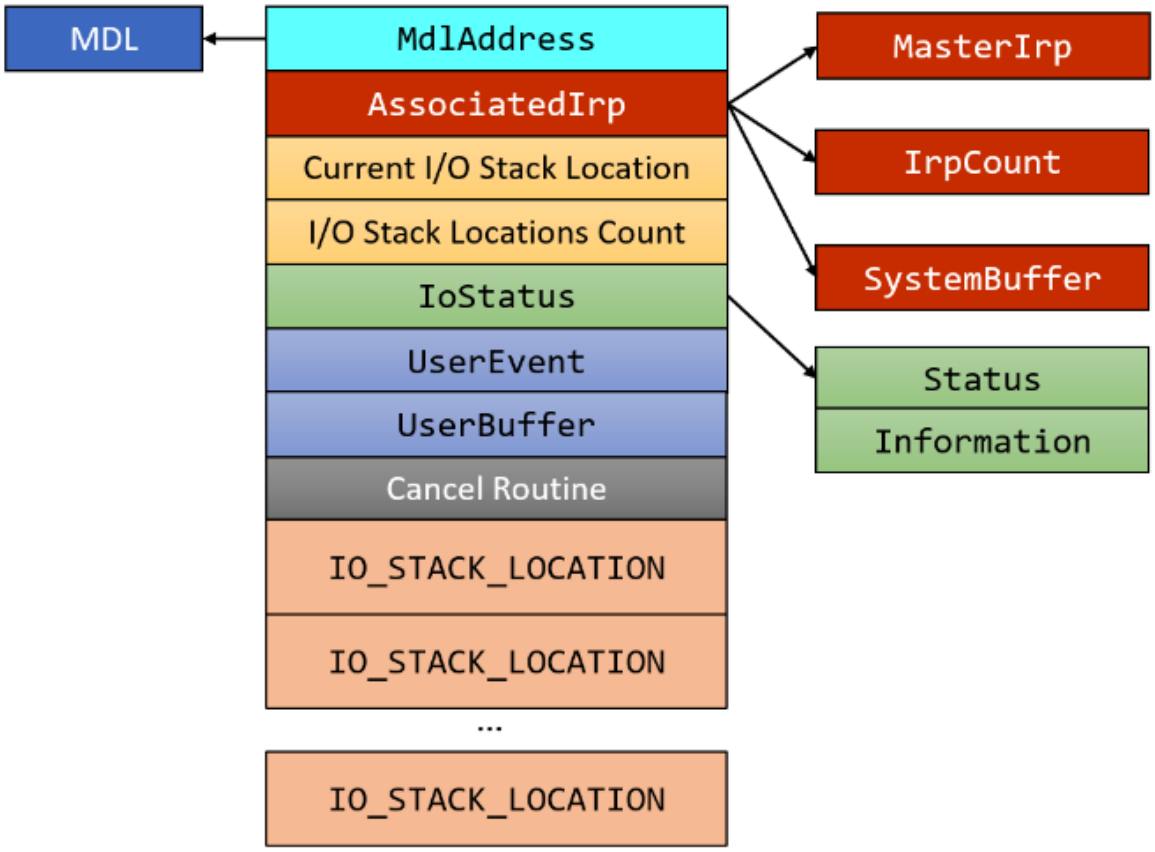
3. Комбинация (1) и (2) - драйвер может проверить IRP, сделать что-нибудь (например, зарегистрировать запрос), а затем передать его. Или он может внести некоторые изменения в расположение следующего стека ввода-вывода и затем передайть запрос.

4. Передача запроса и получения уведомления, когда запрос завершится устройством нижнего уровня — Любой слой (кроме самого нижнего) может настроить процедуру завершения ввода-вывода, вызвав IoSetCompletion.

5. Запуск асинхронной обработки IRP - драйвер может захотеть обработать запрос, но если запрос длинный (типично для аппаратного драйвера, но также может быть в случае программного драйвера), драйвер может пометить IRP как ожидающий, вызвав IoMarkIrpPending и вернув STATUS_PENDING от его режима отправки. В конце концов, ему придется завершить IRP.

Как только какой-то уровень вызывает IoCompleteRequest, IRP разворачивается и начинает «подниматься» обратно к создателю IRP (обычно к Менеджерам). Если процедуры завершения были зарегистрированы, они будут вызываться в порядке, обратном регистрации, то есть снизу вверх.

IRP и расположение стека ввода-вывода



Вот краткое изложение полей на рисунке:

• IoStatus - содержит статус (NT_STATUS) IRP и информационное поле. Информационное поле является полиморфным, имеет тип ULONG_PTR (32- или 64-битное целое число), но его значение зависит от типа IRP. Например, для пакетов IRP для чтения и записи его значение количество байтов, переданных в операции.

• UserBuffer - содержит указатель на буфер пользователя для соответствующих IRP.

• UserEvent - это указатель на объект события (KEVENT), который был предоставлен клиентом, если вызов является асинхронным, и такое событие было предоставлено.

• AssociatedIrp - это объединение состоит из трех членов, только один (не более) из которых действителен:

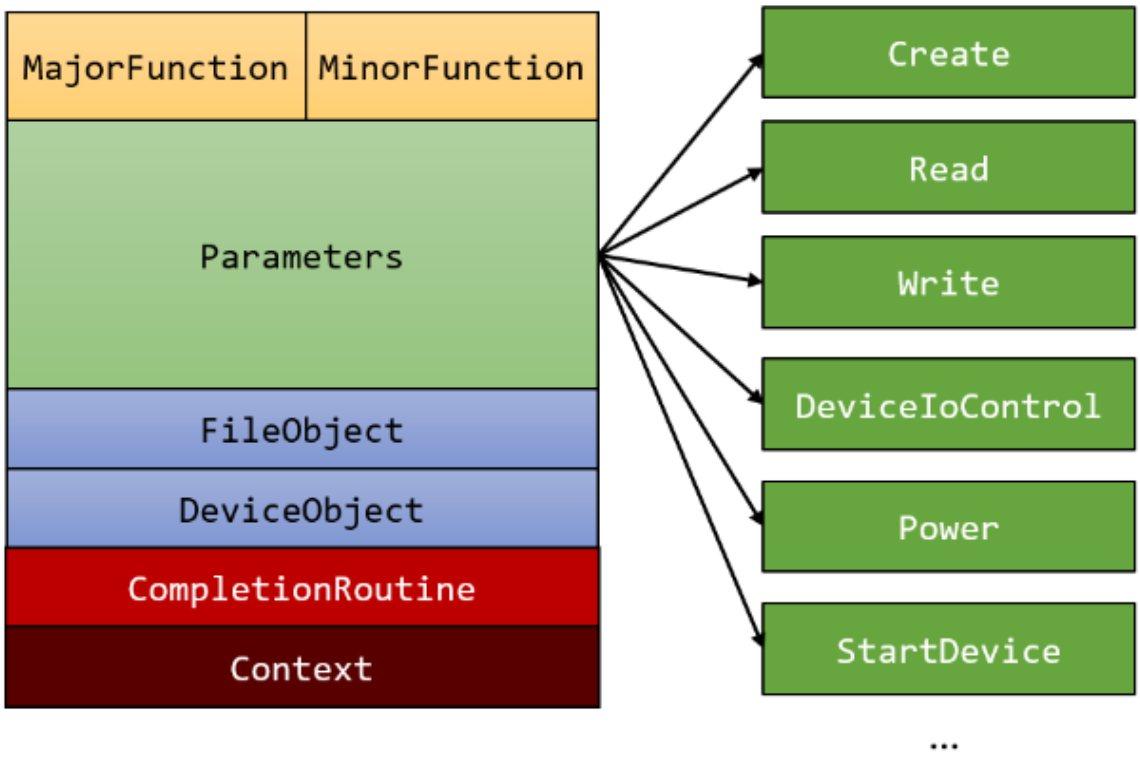
- SystemBuffer - наиболее часто используемый член. Это указатель на выделенный системой невыгружаемый буфер пула, используемый для операций буферизованного ввода-вывода.
- MasterIrp - указатель на «главный» пакет IRP, если этот пакет IRP является связанным пакетом IRP.
- IrpCount - для самого главного IRP в этом поле указывается количество связанных IRP.

• Cancel Routine - указатель на подпрограмму отмены, которая вызывается (если не NULL), если запрашивается операция отмена, например, с функциями пользовательского режима CancelIo и CancelIoEx.

Программные драйверы редко нуждаются в процедурах отмены, поэтому мы не будем использовать их в этой книге.

• MdlAddress - указывает на необязательный список дескрипторов памяти (MDL). MDL - это данные ядра, структура, которая знает, как описать буфер в ОЗУ.

Каждый IRP сопровождается одним или несколькими IO_STACK_LOCATION. На следующем рисунке показаны важные поля в IO_STACK_LOCATION.



Вот краткое изложение полей, показанных на рисунке:
• MajorFunction - это основная функция IRP (IRP_MJ_CREATE, IRP_MJ_READ и т. д.). Это поле иногда полезно, если драйвер указывает более одного кода основной функции на те-же процедуры обращения.
• MinorFunction - некоторые типы IRP имеют второстепенные функции. Это IRP_MJ_PNP, IRP_MJ_POWER и IRP_MJ_SYSTEM_CONTROL (WMI). Типичный код для этих обработчиков имеет переключатель основанный на MinorFunction. Мы не будем использовать эти типы IRP в этой книге, за исключением драйверов фильтров для аппаратных устройств, которые мы рассмотрим подробно в главе 11.
• FileObject - FILE_OBJECT, связанный с этим IRP. В большинстве случаев не требуется, но есть и доступен для подпрограмм отправки.
• DeviceObject - объект устройства, связанный с этим IRP. Процедуры диспетчеризации получают указатель, поэтому обычно доступ к этому полю не требуется.
• CompletionRoutine - процедура завершения, которая устанавливается для предыдущего (верхнего) уровня (установлен с IoSetCompletionRoutine).
• Контекст - аргумент, передаваемый в процедуру завершения (если есть).
• Параметры - это чудовищное объединение содержит несколько структур, каждая из которых действительна для определенной операции. Например, в операции чтения (IRP_MJ_READ) структуру Parameters.Read следует использовать для получения дополнительной информации об операции чтения.

Текущее расположение стека ввода-вывода, полученное с помощью IoGetCurrentIrpStackLocation, содержит большую часть

параметры запроса в объединении. Драйвер должен получить доступ к правильной структуре, как мы уже видели в главе 4 и еще раз увидим в этой и последующих главах.

Просмотр информации IRP

При отладке или анализе дампов ядра несколько команд могут быть полезны для поиска или изучения IRP.

Команду! Irpfind можно использовать для поиска IRP - всех IRP или IRP, соответствующих определенным критериям.

Использование! Irpfind без каких-либо аргументов выполняет поиск всех пакетов IRP в невыгружаемом пуле (ax).

Вот пример некоторого вывода при поиске всех IRP:

```
lkd> !irpfind
Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool

Scanning large pool allocation table for tag 0x3f707249 (Irp?) (ffffbf0a87610000 : f\
fffbf0a87910000)

  Irp            [ Thread ]         irpStack: (Mj,Mn)   DevObj          [Driver]    \
     MDL Process
ffffbf0aa795ca30 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a9a8ef010 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a8e68ea20 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a90deb710 [ffffbf0a808a1080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a99d1da90 [0000000000000000] Irp is complete (CurrentLocation 10 > StackCount\
 9)
ffffbf0a74cec940 [0000000000000000] Irp is complete (CurrentLocation 8 > StackCount \
7)
ffffbf0aa0640a20 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a89acf4e0 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]
ffffbf0a89acfa50 [ffffbf0a7fcde080] irpStack: ( c, 2)  ffffbf0a74d20050 [ \FileSyste\
m\Ntfs]

(truncated)
```

Столкнувшись с конкретным IRP, команда! Irp проверяет IRP, обеспечивая хороший обзор его данные. Как всегда, команду dt можно использовать с типом _IRP для просмотра всей структуры IRP.

Вот пример одного IRP, просматриваемого с помощью! Irp:

```
kd> !irp ffffbf0a8bbada20
Irp is active with 13 stacks 12 is current (= 0xffffbf0a8bbade08)
 No Mdl: No System Buffer: Thread ffffbf0a7fcde080:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [N/A(0), N/A(0)]
    0  0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
    0  0 00000000 00000000 00000000-00000000

(truncated)

    Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
    0  0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
>[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
    0 e1 ffffbf0a74d20050 ffffbf0a7f52f790 fffff8015c0b50a0-ffffbf0a91d99010 Success\
 Error Cancel pending
       \FileSystem\Ntfs
        Args: 00004000 00000051 00000000 00000000
 [IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
    0  0 ffffbf0a60e83dc0 ffffbf0a7f52f790 00000000-00000000
       \FileSystem\FltMgr
    Args: 00004000 00000051 00000000 00000000
```

Команды! Irp выводят список местоположений стека ввода-вывода и хранящуюся в них информацию.

Процедуры отправки

Мы уже видели в главе 4, что одним из важных аспектов DriverEntry является настройка отправки. Поле majorFunction в DRIVER_OBJECT - Это массив указателей на функции, индекс соответствует основной функции.

Все процедуры отправки имеют один и тот же прототип, повторенный здесь для удобства с использованием DRIVER_DISPATCH typedef из WDK (несколько упрощено для ясности):

```
typedef NTSTATUS DRIVER_DISPATCH (
            _In_ PDEVICE_OBJECT DeviceObject,
            _Inout_ PIRP Irp);
```

Соответствующая процедура диспетчеризации (основанная на коде основной функции) - это первая процедура в драйвере, которая видит запрос. Обычно он вызывается контекстом запрашивающего потока, то есть потоком, который вызвал соответствующий API (например, ReadFile) в IRQL PASSIVE_LEVEL (0).

Однако возможно, что драйвер фильтра сидя на вершине этого устройства, отправил запрос в другом контексте - это может быть какой-то другой поток, не связанный с исходным запросом, и даже с более высоким IRQL, таким как DISPATCH_LEVEL (2).

Мы обсудим, как правильно разрешить эту ситуацию в раздел «Доступ к пользовательским буферам» далее в этой главе.

Все процедуры диспетчеризации следуют определенному набору операций:

1. Проверка на наличие ошибок. Процедура отправки обычно сначала проверяет наличие логических ошибок, если это возможно. Например, операции чтения и записи содержат буферы - имеют ли эти буферы соответствующие размеры ?

Для DeviceIoControl есть управляющий код в дополнение к потенциально двум буферам. Драйверу необходимо убедиться, что управляющий код распознается. Если обнаружена какая-либо ошибка, IRP завершается немедленно с соответствующим статусом.

2. Обработка запроса соответствующим образом.

Вот список наиболее распространенных процедур диспетчеризации программного драйвера:

• IRP_MJ_CREATE - соответствует вызову CreateFile из пользовательского режима или ZwCreateFile в режим ядра. Эта основная функция по сути является обязательной, иначе ни один клиент не сможет открыть дескриптор устройства, управляемого этим драйвером.

• IRP_MJ_CLOSE - противоположность IRP_MJ_CREATE. Вызывается CloseHandle из пользовательского режима или ZwClose из режима ядра, когда последний дескриптор файлового объекта собирается закрыть.

• IRP_MJ_READ - соответствует операции чтения, обычно вызываемой из пользовательского режима ReadFile или режим ядра с ZwReadFile.

• IRP_MJ_WRITE - соответствует операции записи, обычно вызываемой из пользовательского режима WriteFile или режим ядра с ZwWriteFile.

• IRP_MJ_DEVICE_CONTROL - соответствует вызову DeviceIoControl из пользовательского режима или ZwDeviceIoControlFile из режима ядра (в ядре есть другие API, которые могут генерировать IRP_MJ_DEVICE_CONTROL IRP).

• IRP_MJ_INTERNAL_DEVICE_CONTROL - аналогично IRP_MJ_DEVICE_CONTROL, но только для вызывающих ядер.

Завершение запроса

Как только драйвер решает обработать IRP (это означает, что он не передается другому драйверу), он должен в итоге это сделать. В противном случае у нас будет утечка наших дескрипторах - запрашивающий поток не может завершится, что приведет к «зомби процессам".

Завершение запроса означает вызов IoCompleteRequest после заполнения статуса запроса.

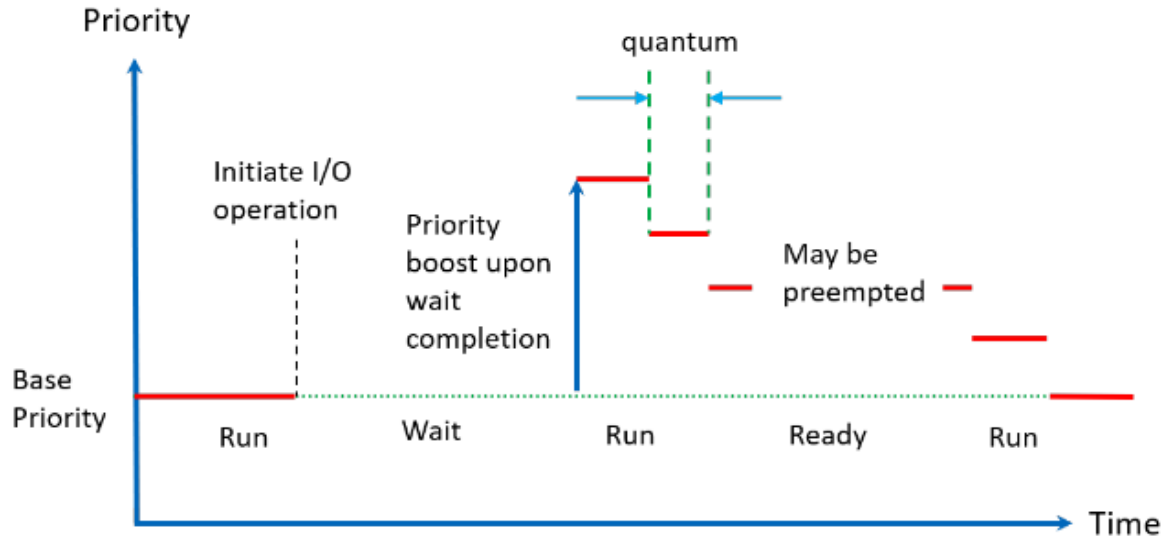
В следующем фрагменте кода показано, как выполнить запрос в процедуре отправки:

```
NTSTATUS MyDispatchRoutine(PDEVICE_OBJECT, PIRP Irp) {
	//...
	Irp->IoStatus.Status = STATUS_XXX;
	Irp->IoStatus.Information = NumberOfBytesTransfered;
	pe
	IoCompleteRequest(Irp, IO_NO_INCREMENT);
	return STATUS_XXX;
}
```

IoCompleteRequest принимает два аргумента: сам IRP и необязательное значение для временного увеличения приоритета исходного потока (поток, который инициировал запрос в первую очередь). В в большинстве случаев для программных драйверов рассматриваемый поток является исполняющим потоком, поэтому увеличивать приоритет потока не нужно. Значение IO_NO_INCREMENT определено как ноль, поэтому в приведенном выше коде нет увеличения.

Однако драйвер может решить дать потоку увеличение, независимо от того, является ли он вызывающим потоком или нет. В этом случае приоритет потока перескакивает с заданным увеличением, и затем разрешается выполнение одного кванта с этим новым приоритетом до того, как приоритет уменьшится на единицу, затем он может получить другой квант с пониженным приоритетом и так далее, пока его приоритет не вернется к исходному уровню.

Следующий рисунок иллюстрирует этот сценарий.



Доступ к пользовательским буферам

Данная процедура диспетчеризации первой видит IRP. Некоторые процедуры отправки, в основном IRP_MJ_READ, IRP_MJ_WRITE и IRP_MJ_DEVICE_CONTROL принимают буферы, предоставленные клиентом - в большинстве случаев из пользовательского режима.

Обычно процедура отправки вызывается в IRQL 0 и в контексте запрашивающего потока,

что означает, что указатели буферов, предоставленные пользовательским режимом, легко доступны: IRQL равен 0, поэтому сбои страниц обрабатываются нормально, и поток является инициатором запроса, поэтому указатели действительны в этом контексте процесса.

Однако могут быть проблемы. Как мы видели в главе 6, даже в этом удобном контексте

(запрашивая поток и IRQL 0), другой поток в клиентском процессе может освободить

переданный буфер (буферы), прежде чем драйвер получит возможность проверить их, что приведет к нарушению доступа.

Решение, которое мы использовали в главе 6, - использовать блок __try / __except для обработки любого доступа.

В некоторых случаях даже этого недостаточно. Например, если у нас есть код, работающий на IRQL 2 (например, как DPC, запущенный в результате истечения таймера), мы не можем безопасно получить доступ к пользовательским буферам в этом контексте.

Здесь есть две проблемы:

• IRQL равен 2, что означает невозможность обработки ошибок страницы.

• Поток, выполняющий DPC, является произвольным, поэтому сам указатель не имеет значения, какой бы процесс ни был текущим на этом процессоре.

Использование обработки исключений в таком случае не будет работать правильно, потому что мы будем получать доступ к некоторым ячейкам памяти, которая по существу недопустима в контексте этого случайного процесса.

Даже если доступ успешен (потому что эта память выделяется в этом случайном процессе), мы будем обращаться к произвольной памяти, а не к исходному буферу, предоставленному для запроса.

Все это означает, что должен быть какой-то способ получить доступ к исходному пользовательскому буферу в неудобном контексте. Фактически, ядро предоставляет для этой цели два способа: Буферизованный ввод-вывод и прямой ввод-вывод. В следующих разделах мы увидим, что означает каждая из этих схем и как их использовать.

Буферизованный ввод/вывод

Буферизованный ввод-вывод - самый простой из двух способов. Чтобы получить поддержку буферизованного ввода-вывода для чтения и записи, на объекте устройства должен быть установлен такой флаг:

```
DeviceObject->Flags |= DO_BUFFERED_IO; // DO = Device Object
```

Информацию о буферах IRP_MJ_DEVICE_CONTROL см. В разделе «Пользовательские буферы для IRP_MJ_DEVICE_CONTROL» далее в этой главе.

Вот шаги, предпринимаемые диспетчером ввода-вывода и драйвером при поступлении запроса на чтение или запись:

1. Диспетчер ввода-вывода выделяет буфер из невыгружаемого пула того же размера, что и пользовательский буфер. Он хранит указатель на этот новый буфер в члене AssociatedIrp-> SystemBuffer IRP. (Размер буфера можно найти в текущем местоположении стека ввода-вывода Parameters.Read.Length или Parameters.Write.Length.)

2. Для запроса на запись диспетчер ввода-вывода копирует буфер пользователя в системный буфер.

3. Только теперь вызывается диспетчерская программа драйвера. Драйвер может использовать указатель системного буфера напрямую без каких-либо проверок, потому что буфер находится в системном пространстве (его адрес абсолютный - то же самое для любого контекста процесса) и в любом IRQL, потому что буфер выделяется из невыгружаемого пула, поэтому его нельзя выгружать.

4. Как только драйвер завершит IRP (IoCompleteRequest), диспетчер ввода-вывода (для запросов на чтение) копирует системный буфер обратно в буфер пользователя (размер копии определяется в IoStatus.Information в IRP, установленном драйвером).

5. Наконец, диспетчер ввода-вывода освобождает системный буфер.

На следующих рисунках показаны шаги, предпринятые с буферизованным вводом-выводом.

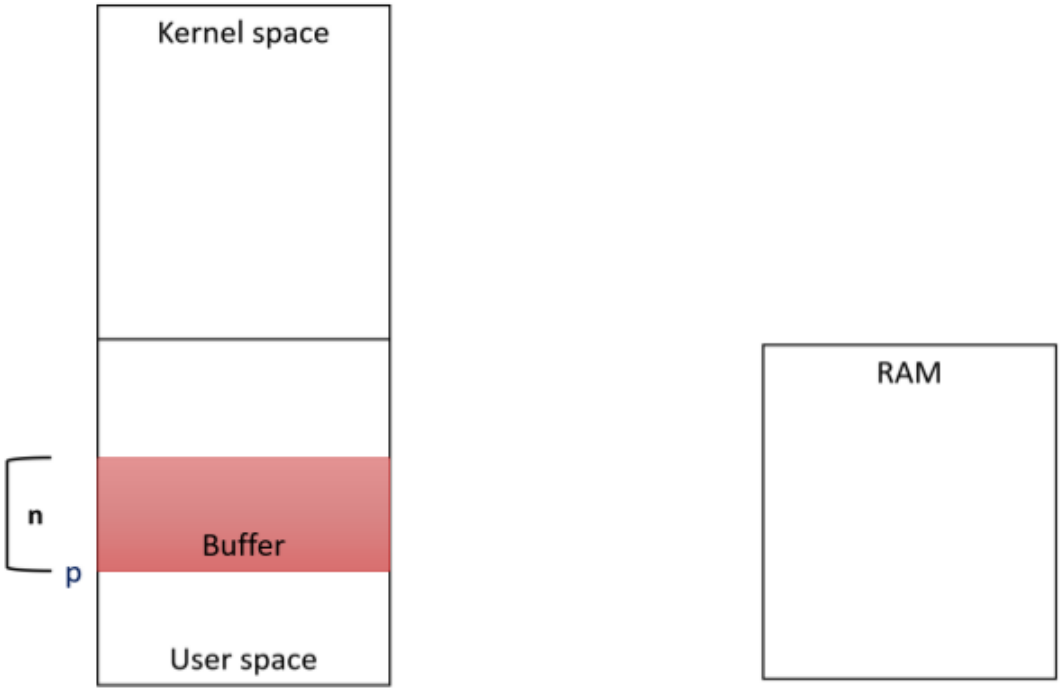


Figure 7-8a: Buffered I/O: initial state

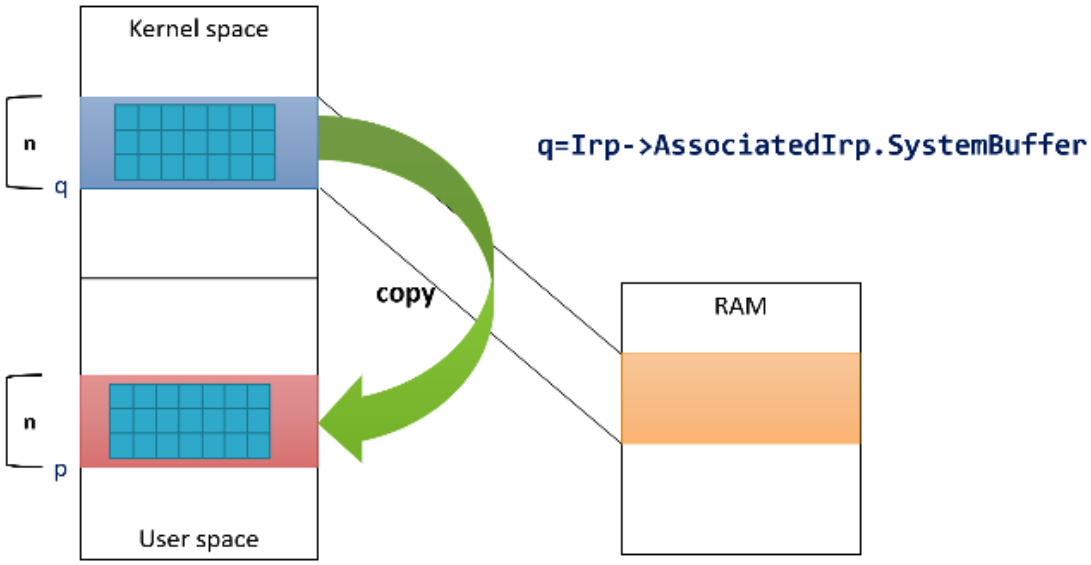


Figure 7-8d: Buffered I/O: on IRP completion, I/O manager copies buffer back (for read)

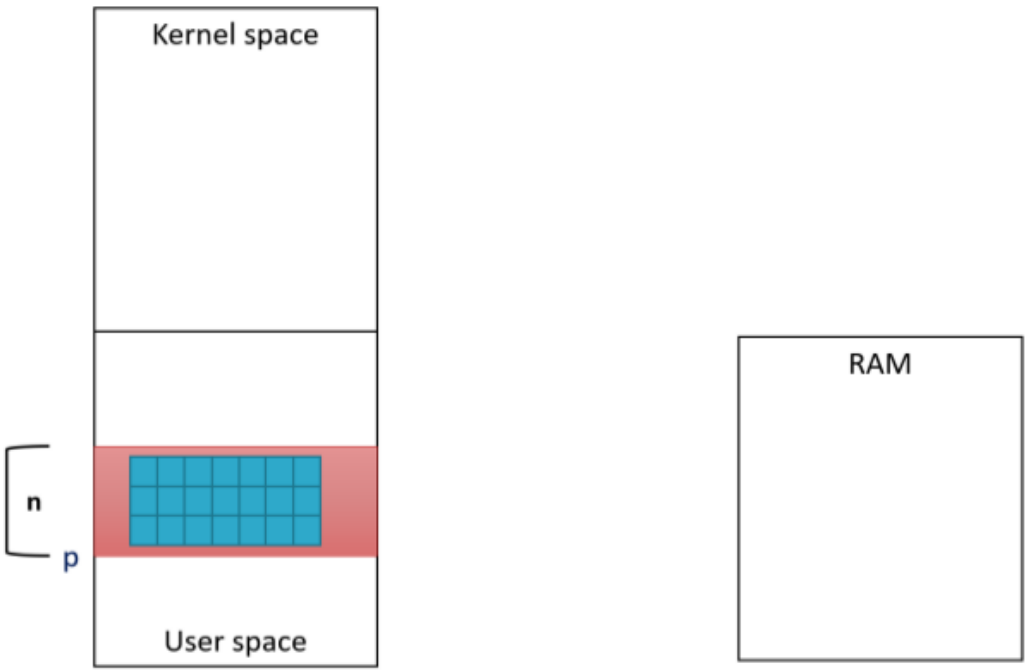


Figure 7-8e: Buffered I/O: final state - I/O manager frees system buffer

Буферизованный ввод-вывод имеет следующие характеристики:

• Простота использования - просто укажите флаг в объекте устройства, а обо всем остальном позаботится диспетчер ввода-вывода.

• Он всегда включает копию, что означает, что его лучше всего использовать для небольших буферов (обычно до одной страницы). Копирование больших буферов может быть дорогостоящим. В этом случае другой вариант, прямого I/O, должен использоваться вместо этого.

Прямой ввод/вывод

Цель прямого ввода-вывода - разрешить доступ к пользовательскому буферу в любом IRQL и любом потоке, но без его копирования.

Для запросов на чтение и запись выбор прямого ввода-вывода выполняется с другим флагом объекта устройства:

DeviceObject-> Flags | = DO_DIRECT_IO;

Как и в случае с буферизованным вводом-выводом, этот выбор влияет только на запросы чтения и записи..

Вот шаги, необходимые для обработки прямого ввода-вывода:

1. Диспетчер ввода-вывода сначала проверяет правильность пользовательского буфера, а затем переводит его в физический адрес памяти.

2. Затем он блокирует буфер в памяти, поэтому его нельзя выгружать до дальнейшего уведомления. Это решает одну из проблем с доступом к буферу - ошибки страницы не могут произойти, поэтому доступ к буферу в любом IRQL безопасен.

3. Диспетчер ввода-вывода создает список дескрипторов памяти (MDL), структуру данных, которая знает, как буфер отображается в ОЗУ. Адрес этой структуры данных хранится в поле MdlAddress IRP.

4. На этом этапе драйвер получает вызов своей процедуры диспетчеризации. Буфер пользователя заблокирован в ОЗУ, недоступен из произвольного потока. Когда драйвер требует доступ к буферу, он должен вызывать функцию, которая отображает тот же пользовательский буфер в систему ,адрес который по определению действителен в любом контексте процесса. Таким образом, мы получаем два сопоставления с тем же буфером. Один взят с исходного адреса (действителен только в контексте процесса инициатора запроса), а другой - в системном пространстве.

API для вызова - это MmGetSystemAddressForMdlSafe, передающий MDL, созданный диспетчером ввода-вывода. Возвращаемое значение - системный адрес.

5. После того, как драйвер завершит запрос, диспетчер ввода-вывода удаляет второе сопоставление (системное), освобождает MDL и разблокирует буфер пользователя, поэтому его можно выгружать как обычно.

На следующих рисунках показаны шаги, предпринятые с прямым вводом-выводом.

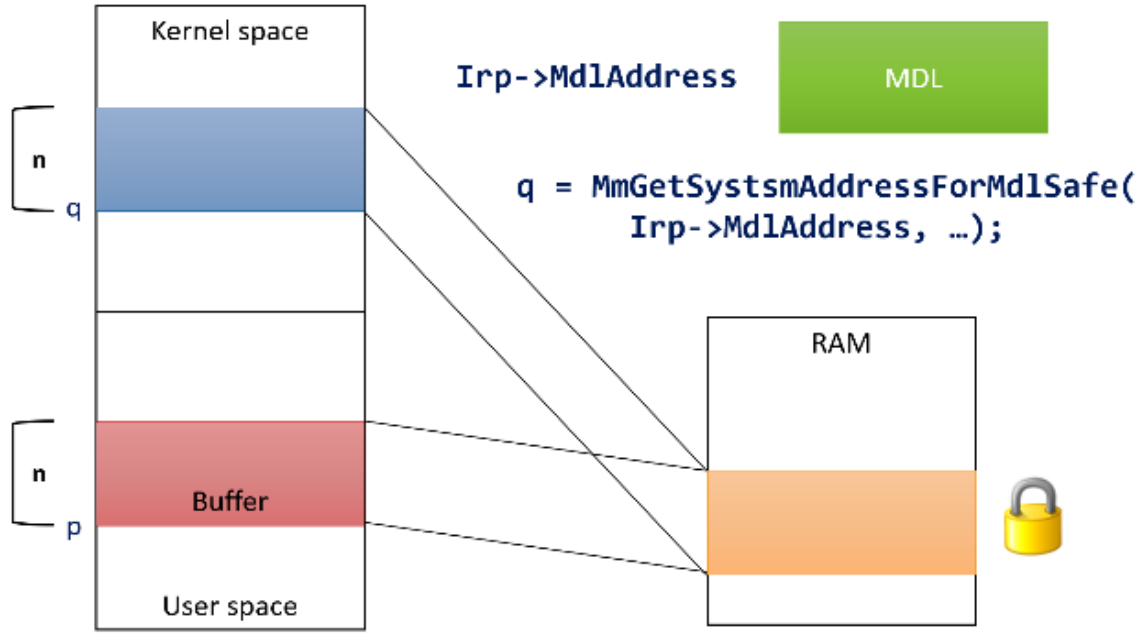


Figure 7-9d: Direct I/O: the driver double-maps the buffer to a system address

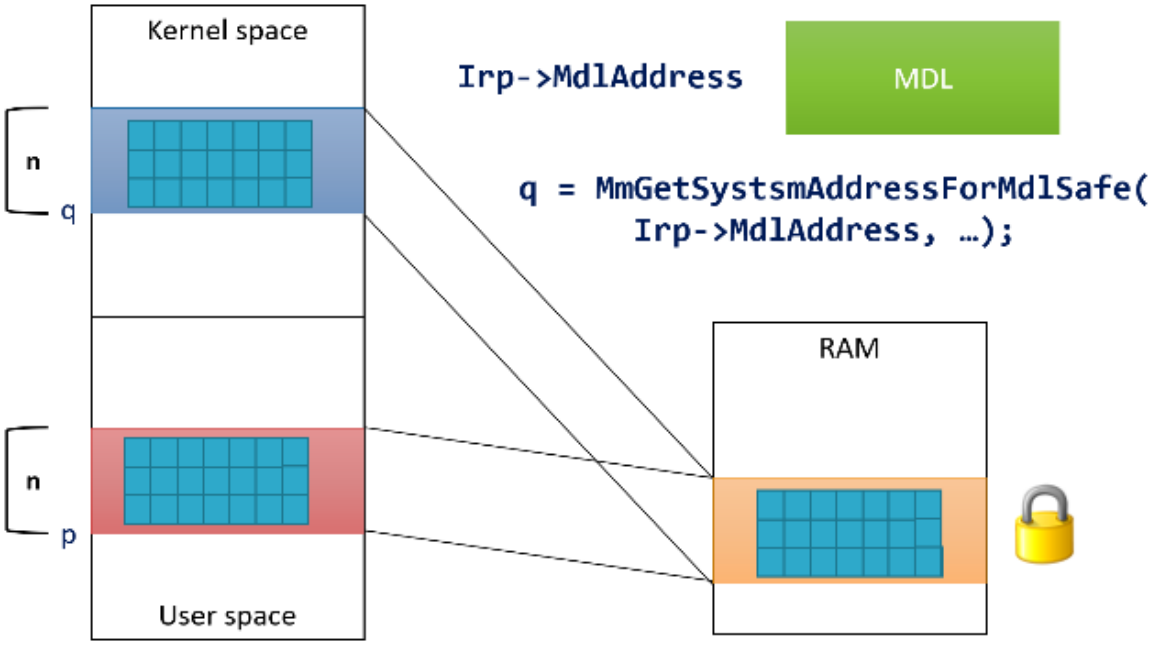


Figure 7-9e: Direct I/O: the driver accesses the buffer using the system address

Обратите внимание, что копирования нет вообще. Драйвер просто читает/записывает в буфер пользователя напрямую, используя системный адрес.

MmGetSystemAddressForMdlSafe принимает MDL и приоритет страницы (MM_PAGE_PRIORITY). Большинство драйверов указывают NormalPagePriority, но есть также LowPagePriority и HighPagePriority. Этот приоритет указывает системе на важность отображения.

Если MmGetSystemAddressForMdlSafe не отработал корректно, он возвращает NULL.

Это должно быть редким явлением, но может произойти в условиях нехватки памяти. Драйвер должен это проверить.

Если возвращается NULL, драйвер должен завершить IRP со статусом STATUS_INSUFFICIENT_RESOURCES .

Пользовательские буферы для IRP_MJ_DEVICE_CONTROL

В последних двух разделах обсуждались буферизованный ввод-вывод и прямой ввод-вывод, поскольку они относятся к запросам на чтение и запись. Для IRP_MJ_DEVICE_CONTROL метод доступа к буферизации предоставляется на основе кода управления. Как напоминаем, что это прототип функции пользовательского режима DeviceIoControl (он похож на ZwDeviceIoControlFile):

```
BOOL DeviceIoControl(
    HANDLE hDevice,              // handle to device or file
    DWORD dwIoControlCode,       // IOCTL code (see <winioctl.h>)
    PVOID lpInBuffer,            // input buffer
    DWORD nInBufferSize,         // size of input buffer
    PVOID lpOutBuffer,           // output buffer
    DWORD nOutBufferSize,        // size of output buffer
    PDWORD lpdwBytesReturned,    // # of bytes actually returned
    LPOVERLAPPED lpOverlapped); // for async. operation
```

Здесь есть три аргумента: код управления вводом-выводом и два необязательных буфера, обозначенных как «ввод». и «вывод». Как оказалось, способ доступа к этим буферам зависит от управляющего кода, что очень удобно, потому что разные запросы могут иметь разные требования, связанные с доступом буферов пользователя. В главе 4 мы уже видели, что управляющий код состоит из четырех аргументов, предоставляемых макросом CTL_CODE, повторенный здесь для удобства:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
      ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

Третий аргумент (Метод) - это ключ к выбору метода буферизации для доступа к вводу.

Вот варианты:

• METHOD_NEITHER - это значение означает, что от диспетчера ввода-вывода не требуется помощь, поэтому драйвер будет работать с буферами самостоятельно. Это может быть полезно, например, если конкретный код не требует буфера - вся необходимая информация содержится в управляющем коде - лучше всего сообщите менеджеру ввода-вывода, что ему не нужно выполнять никаких дополнительных действий.В этом случае указатель на буфер ввода пользователя сохраняется в текущем местоположении стека ввода-вывода.

• METHOD_BUFFERED - это значение указывает буферизованный ввод-вывод как для входного, так и для выходного буфера. При запуске запроса диспетчер ввода-вывода выделяет системный буфер из невыгружаемого пула. с размером, который является максимальной длиной буферов ввода и вывода. Затем он копирует входной буфер в системный буфер. Только теперь процедура отправки IRP_MJ_DEVICE_CONTROL вызывается. Когда запрос завершается, диспетчер ввода-вывода копирует указанное количество байтов. с полем IoStatus.Information в IRP в буфер вывода пользователя.

• METHOD_IN_DIRECT и METHOD_OUT_DIRECT - Оба эти значения означает то же самое, что касается методов буферизации: входной буфер использует буферезированный ввод-вывод, а выходной буфер используют прямой ввод-вывод. Единственная разница между этими двумя значениями - может ли буфер вывода быть прочитан (METHOD_IN_DIRECT) или записан (METHOD_OUT_DIRECT).

Собираем все вместе: нулевой драйвер

В этом разделе мы воспользуемся тем, что узнали в этой (и ранее) главе, и создадим драйвер и клиентское приложение. Драйвер называется Zero и имеет следующие характеристики:

• Для запросов на чтение он обнуляет предоставленный буфер.

• Для запросов на запись он просто использует предоставленный буфер, аналогично классическому нулевому устройству.

Драйвер будет использовать Direct I/O, чтобы не нести накладные расходы на копирование, поскольку буферы, предоставляемые клиентом потенциально может быть очень большим.

Мы начнем проект с создания «Пустого проекта WDM» в Visual Studio и присвоения ему именя Zero. Затем мы удалим созданный файл INF.

Использование предварительно скомпилированного заголовка

Один метод, который мы можем использовать, который не является специфическим для разработки драйверов, но в целом полезен, использует предварительно скомпилированные заголовки. Предварительно скомпилированные заголовки - это функция Visual Studio, которая помогает ускорить время компиляции.

Предварительно скомпилированный заголовок - это файл заголовка, в котором есть операторы #include для заголовков, которые редко меняются, например, ntddk.h для драйверов.

Предварительно скомпилированный заголовок компилируется один раз и сохраняется в внутренний двоичный формат и используется в последующих компиляциях, которые становятся значительно быстрее.

Выполните следующие действия, чтобы создать и использовать предварительно скомпилированный заголовок:

• Добавьте в проект новый файл заголовка и назовите его pch.h. Этот файл будет служить предварительно скомпилированным заголовок. Добавьте сюда все редко меняющиеся #include:

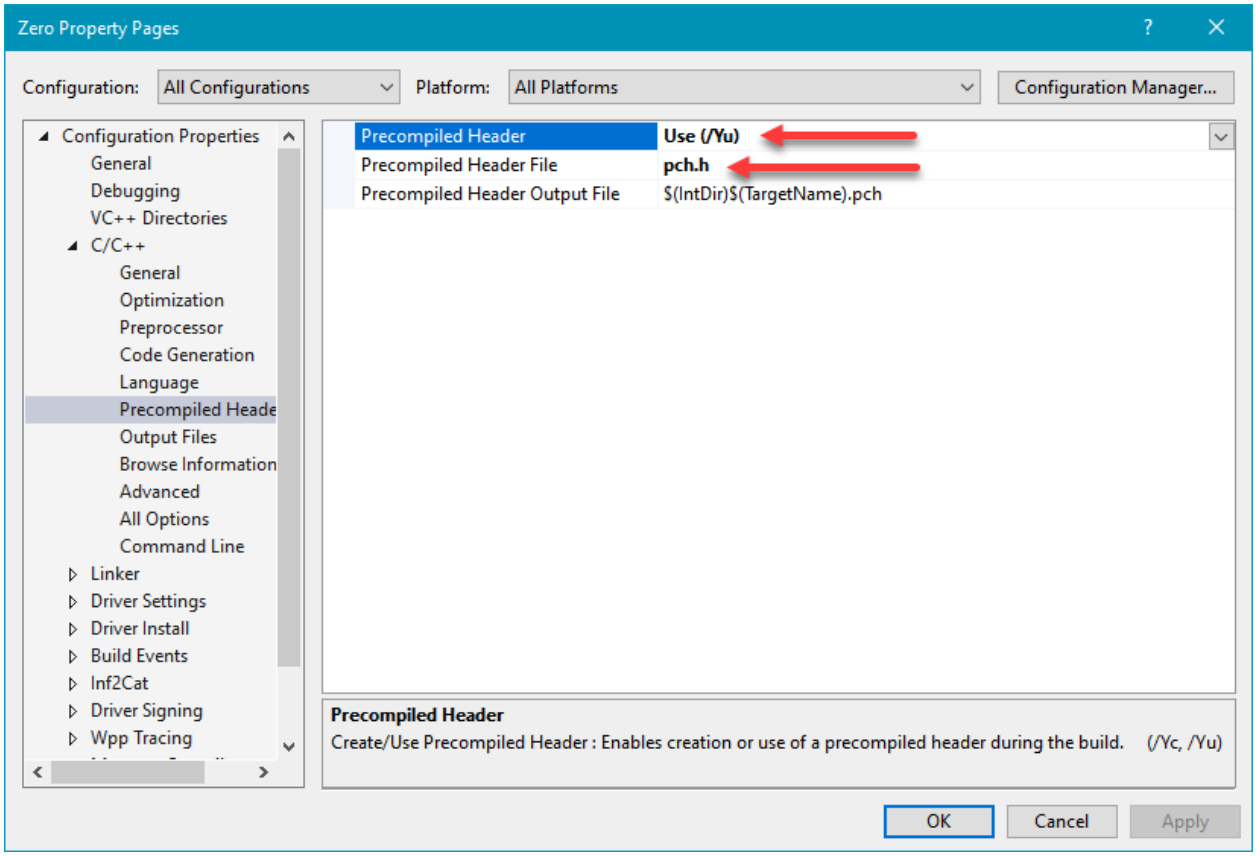
#pragma once

#include <ntddk.h>

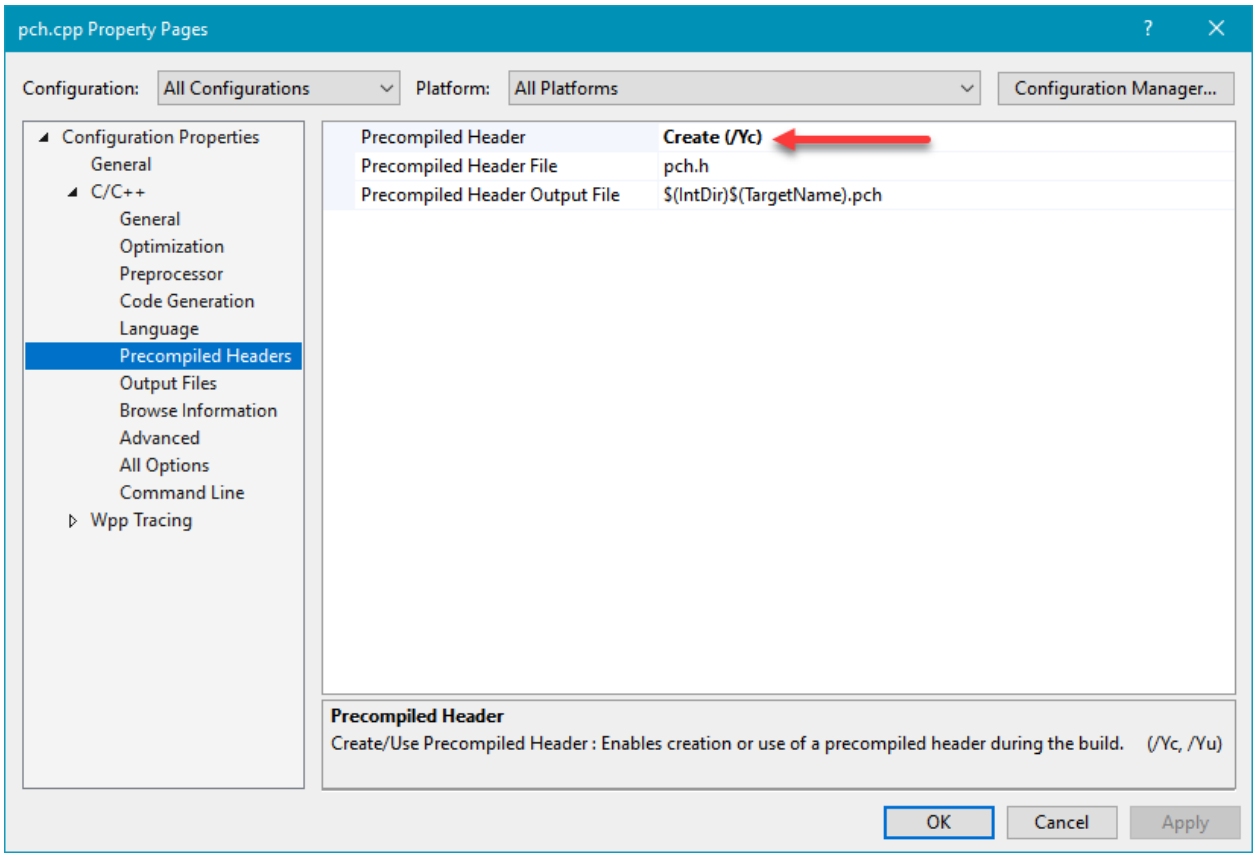
• Добавьте исходный файл с именем pch.cpp и поместите в него единственный #include: сам предварительно скомпилированный заголовок:

#include "pch.h"

• А теперь самое сложное. Сообщаем компилятору, что pch.h - это предварительно скомпилированный заголовок. Откройте свойства проекта, выберите Все конфигурации. Чтобы вам не приходилось настраивать каждую конфигурацию/платформу отдельно, перейдите к C/C ++/Precompiled Headers и установите для Precompiled Header значение «Use», а имя файла - «pch.h». (см. рисунок). Нажмите ОК и, чтобы закрыть диалоговое окно.



Файл pch.cpp должен быть установлен как создатель предварительно скомпилированного заголовка. Щелкните этот файл правой кнопкой мыши в обозревателе решений и выберите Свойства. Перейдите к C/C ++/Precompiled Headers и установите Предварительно скомпилированный заголовок «Create» (см. рисунок). Щелкните OK, чтобы принять настройку.



С этого момента каждый файл C/CPP файл в проекте должен включать #include "pch.h" в первую очередь. Без этого включения проект не скомпилируется.

Процедура DriverEntry

Процедура DriveEntry для драйвера Zero очень похожа на ту, которую мы создали для драйвера в глава 4. Однако в драйвере главы 4 код отменяет любую операцию, которая уже была сделана в случае, если была ошибка.

У нас было всего две операции, которые можно было отменить: создание объекта устройства. и создание символьной ссылки.

Драйвер Zero аналогичен, но мы создадим более надежный и менее подверженный ошибкам код для обработки ошибок во время инициализации. Начнем с настройки процедуры выгрузки и процедуры отправки:

```
#define DRIVER_PREFIX "Zero: "
// DriverEntry
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
	UNREFERENCED_PARAMETER(RegistryPath);
	DriverObject->DriverUnload = ZeroUnload;
	DriverObject->MajorFunction[IRP_MJ_CREATE] = DriverObject-
>MajorFunction[IRP_MJ_\
		CLOSE] = ZeroCreateClose;
	DriverObject->MajorFunction[IRP_MJ_READ] = ZeroRead;
	DriverObject->MajorFunction[IRP_MJ_WRITE] = ZeroWrite;
```

Теперь нам нужно создать объект устройства и символическую ссылку и обработать ошибки в более общем и надежный способ.

Уловка, которую мы будем использовать, - это блок do/while (false), который на самом деле не является циклом, но позволяет выйти из блока с помощью простого оператора break на случай, если что-то пойдет не так:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\Zero");
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Zero");
PDEVICE_OBJECT DeviceObject = nullptr;
auto status = STATUS_SUCCESS;
do {
	status = IoCreateDevice(DriverObject, 0, &devName, FILE_DEVICE_UNKNOWN,
		0, FALSE, &DeviceObject);
	if (!NT_SUCCESS(status)) {
		KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n",
			status));
		break;
	}
	// set up Direct I/O
	DeviceObject->Flags |= DO_DIRECT_IO;
	status = IoCreateSymbolicLink(&symLink, &devName);
	if (!NT_SUCCESS(status)) {
```

```
            KdPrint((DRIVER_PREFIX "failed to create symbolic link (0x
%08X)\n",
                        status));
            break;
        }
} while (false);
if (!NT_SUCCESS(status)) {
        if (DeviceObject)
            IoDeleteDevice(DeviceObject);
}
return status;
```

Схема проста: если в любом вызове возникнет ошибка, просто выйдите из «цикла». Вне цикла, проверьте статус и, если это сбой, отмените все выполненные операции. Имея в руках эту схему, легко добавить больше инициализаций (которые нам понадобятся в более сложных драйверах).

Обратите внимание, что мы также инициализируем устройство для использования прямого ввода-вывода для операций чтения и записи.

Процедура отправки чтения

Прежде чем мы перейдем к фактической процедуре отправки чтения, давайте создадим вспомогательную функцию, которая упрощает заполнение IRP с заданным статусом и информацией:

```
NTSTATUS CompleteIrp(PIRP Irp, NTSTATUS status = STATUS_SUCCESS, ULONG_PTR
info = 0)\
{
      Irp->IoStatus.Status = status;
      Irp->IoStatus.Information = info;
      IoCompleteRequest(Irp, 0);
      return status;
}
```

Теперь мы можем приступить к реализации процедуры отправки чтения. Сначала нам нужно проверить длину буфера, чтобы убедиться, что он не равен нулю. Если это так, просто завершите IRP со статусом ошибки:

```
NTSTATUS ZeroRead(PDEVICE_OBJECT, PIRP Irp) {
      auto stack = IoGetCurrentIrpStackLocation(Irp);
      auto len = stack->Parameters.Read.Length;
```

```
    if (len == 0)
        return CompleteIrp(Irp, STATUS_INVALID_BUFFER_SIZE);
```

Обратите внимание, что длина пользовательского буфера определяется структурой Parameters.Read внутри текущего расположение стека ввода-вывода. Мы настроили прямой ввод-вывод, поэтому нам нужно сопоставить заблокированный буфер с системным пространством, используя MmGetSystemAddressForMdlSafe:

```
auto buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
NormalPagePriority);
if (!buffer)
    return CompleteIrp(Irp, STATUS_INSUFFICIENT_RESOURCES);
```

Функциональность, которую нам нужно реализовать, - это обнулить данный буфер. Мы можем использовать простой memset вызов, чтобы заполнить буфер нулями, а затем выполнить запрос:

```
memset(buffer, 0, len);
return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

Важно установить в информационном поле длину буфера. Это указывает клиенту количество байтов, использованных в операции (в предпоследнем аргументе ReadFile). Этот все, что нам нужно для операции чтения.

Процедура отправки записи

Процедура отправки записи еще проще. Все, что ему нужно сделать, это просто заполнить запрос с длиной буфера, предоставляемым клиентом:

```
NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;
    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

Обратите внимание, что мы даже не вызываем MmGetSystemAddressForMdlSafe, поскольку нам не нужно доступ к фактическому буферу. Это также причина того, что этот вызов не выполняется заранее диспетчером ввода-вывода: драйверу может даже не понадобиться, а может быть, он понадобится только в определенных условиях; так что диспетчер ввода-вывода подготавливает все (MDL) и позволяет драйверу решать, когда и нужно ли выполнять фактическое сопоставление.

Тестовое приложение

Мы добавим в решение новый проект консольного приложения для тестирования операций чтения и записи. Вот простой код для проверки этих операций:

```
int Error(const char* msg) {
	printf("%s: error=%d\n", msg, ::GetLastError());
	return 1;
}

int main() {
	HANDLE hDevice = ::CreateFile(L"\\\\.\\Zero", GENERIC_READ | GENERIC_WRITE,
		0, nullptr, OPEN_EXISTING, 0, nullptr);
	if (hDevice == INVALID_HANDLE_VALUE) {
		return Error("failed to open device");
	}
	// test read
	BYTE buffer[64];
	// store some non-zero data
	for (int i = 0; i < sizeof(buffer); ++i)
		buffer[i] = i + 1;
	DWORD bytes;
	BOOL ok = ::ReadFile(hDevice, buffer, sizeof(buffer), &bytes, nullptr);
	if (!ok)
		return Error("failed to read");
	if (bytes != sizeof(buffer))
		printf("Wrong number of bytes\n");
	// check if buffer data sum is zero
	long total = 0;
	for (auto n : buffer)
		total += n;
	if (total != 0)
		printf("Wrong data\n");
	// test write
```

```
	BYTE buffer2[1024];
	// contains junk
	ok = ::WriteFile(hDevice, buffer2, sizeof(buffer2), &bytes, nullptr);
	if (!ok)
		return Error("failed to write");
	if (bytes != sizeof(buffer2))
		printf("Wrong byte count\n");
	::CloseHandle(hDevice);
}
```

Резюме

В этой главе мы узнали, как обрабатывать IRP, с которыми драйверы работают постоянно. Вооружившись этими знаниями, мы можем начать использовать больше функций ядра, в следующей главе.