



ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОПЕРАЦИОННОЙ СИСТЕМЫ УДАРСТВЕННОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
“МОПЕРАЦИОННОЙ СИСТЕМЫ КОВСКИЙ
ГОПЕРАЦИОННОЙ СИСТЕМЫ УДАРСТВЕННЫЙ
УНИВЕРСИТЕТ
ПРИБОРОПЕРАЦИОННОЙ СИСТЕМЫ ТРОЕНИЯ И
ИНФОРМАТИКИ”

**Кафедра “Персональные
компьютеры и сети”**



РОЩИН А.В.

Организация ввода-вывода

Учебное пособие

Часть 3

**Основы написания драйверов уровня ядра для
операционной системы Windows NT5**



**Москва
2009**

АННОТАЦИЯ

Настоящее учебное пособие предназначено для подготовки студентов различных вычислительных специальностей, изучающих операционные системы различного назначения. Для специальности 230101 эти методические указания могут использоваться в курсе "Организация ввода-вывода" и "Системное программное обеспечение" для самостоятельного изучения материала и подготовки к лекциям.

В третьей части пособия описаны методы и способы написания простейших драйверов под Windows NT5.

Автор: проф. Рощин А.В.

Рецензент: проф., к.т.н. Зеленко Г.В.

Научный редактор: проф., д.т.н. Михайлов Б.М.

Работа рассмотрена и одобрена на заседании кафедры ИТ-4
_____ 2009 г.

Зав. кафедрой: проф., д.т.н. Михайлов Б.М. _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 КРАТКИЙ ОБЗОР АРХИТЕКТУРЫ WINDOWS 2000	6
1.1 Режим пользователя и режим ядра	6
1.2 Сервисы	13
1.3 Пример приложения, использующего сервис	16
1.4 Прототип драйвера режима ядра	23
1.5 Контрольные вопросы	28
2 ИСПОЛЬЗОВАНИЕ ПАКЕТА NUMEGA DRIVER STUDIO ДЛЯ НАПИСАНИЯ WDM- ДРАЙВЕРОВ УСТРОЙСТВ	30
2.1 Система классов DriverWorks	33
2.2 Использование Driver Wizard	56
2.3 Компиляция и установка драйвера	68
2.4 Нарращивание функциональных возможностей драйвера	73
2.5 Контрольные вопросы	89
Литература	90

Введение

В настоящее время операционные системы Windows NT5 (Windows 2000/XP и т. д.) являются наиболее популярными, поэтому данное учебное пособие посвящено именно этому типу операционных систем.

Иногда программистам, кроме собственно написания прикладной программы, приходится писать драйвер для какого-либо устройства, которое не поддерживается стандартной конфигурацией системы. Несмотря на большое количество технической литературы, посвященной программированию в операционной системе Windows, проблемы написания драйверов практически не освещаются. Безусловно, этот вопрос достаточно специфичен для программистов общего плана, но для системных программистов эта тема представляет большой интерес. Даже если программисту не придется непосредственно заниматься написанием драйверов, более глубокое понимание функционирования этой подсистемы операционной системы Windows даст более четкое представление о возможностях функционирования операционной системы в целом. Это позволяет трезво оценивать временные возможности стандартных драйверов и при необходимости получения лучших временных показателей проектируемой системы в целом написать свой, возможно менее функциональный, но более быстрый драйвер. Тема драйверов достаточно обширна, и невозможно в рамках одного учебного пособия научить писать драйверы под Windows NT5. Однако, здесь делается попытка последовательно изложить основные понятия данной операционной системы (разумеется, с учетом [6]), а затем рассмотреть несколько простейших примеров драйверов.

Для решения поставленной задачи логично использовать пакет MASM32 v.8.2, а также комплект разработки драйверов (Windows 2000 Driver Development Kit, DDK), который можно бесплатно скачать с сайта

Microsoft (www.microsoft.com/ddk/). В комплект DDK входят документация (папка help), являющаяся наиболее полным источником информации о внутренних структурах данных и внутрисистемных функциях используемых драйверами устройств, включаемые файлы (*.inc) из папки inc, примеры реализаций драйверов устройств (папка src), утилиты (папка tools). Главное, в DDK входит набор библиотечных файлов (*.lib), необходимых при компоновке законченных драйверов. В DDK есть два комплекта этих файлов — checked build и free build, нам необходим free build (папка libfre) для окончательной версии Windows, называемой свободным выпуском. Функции из библиотечных файлов (папка libchk) отладочной версии (checked build) отличаются более строгой проверкой ошибок. Но, увы, не все так хорошо для прямого использования включаемых файлов, содержащих определения прототипов функций, а также необходимых структур, символьных констант и макросов. Есть одна проблема, заключающаяся в том, что все описания в DDK даны из расчета использования языка C, в то время как мы хотим воспользоваться «великим и могучим» ассемблером. Если еще есть утилиты, позволяющие автоматизировать получение inc-файлов для имеющихся lib-файлов, то формирование inc-файлов, содержащих структуры и константы,— это в основном ручной труд. Эту работу приходится делать самостоятельно. Она не из категории творческих, но в Интернете совершенно бесплатно можно найти достаточно полные версии конвертированных для использования с ассемблером inc-файлов, и таким образом можно начать не с нуля [1].

1 КРАТКИЙ ОБЗОР АРХИТЕКТУРЫ WINDOWS 2000

1.1 Режим пользователя и режим ядра

В Windows 2000/XP существует четкое разграничение двух областей в оперативной памяти и режимов процессора для исполняемого кода:

- 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. Чаще всего в системе установлен существенно 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 приво-

доть 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 [2] приведена на рисунке 1.1.

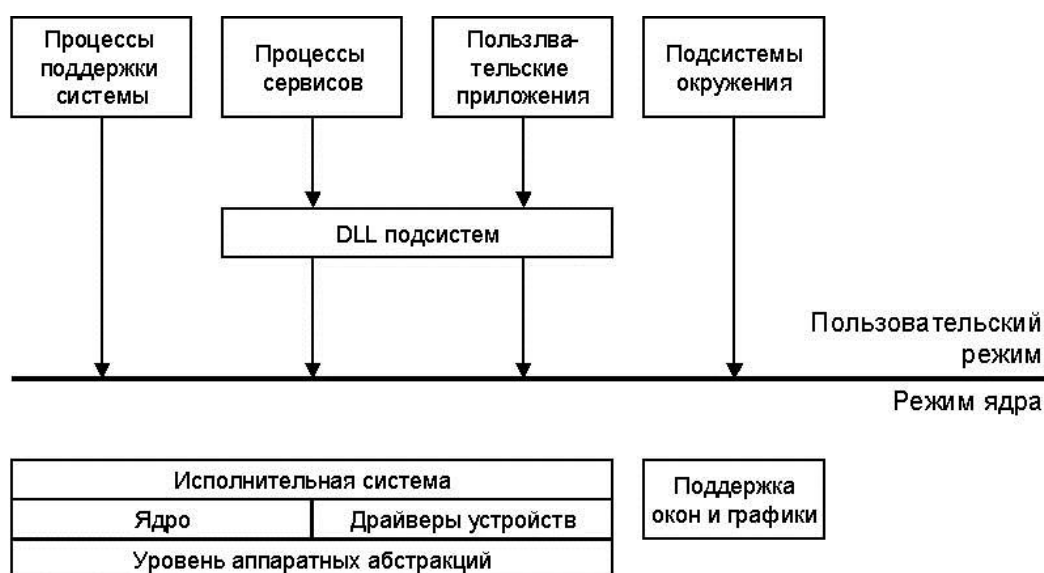


Рисунок 1.1 – Упрощенная схема архитектуры Windows 2000

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 (о сервисах поговорим чуть позже). 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 1.2.

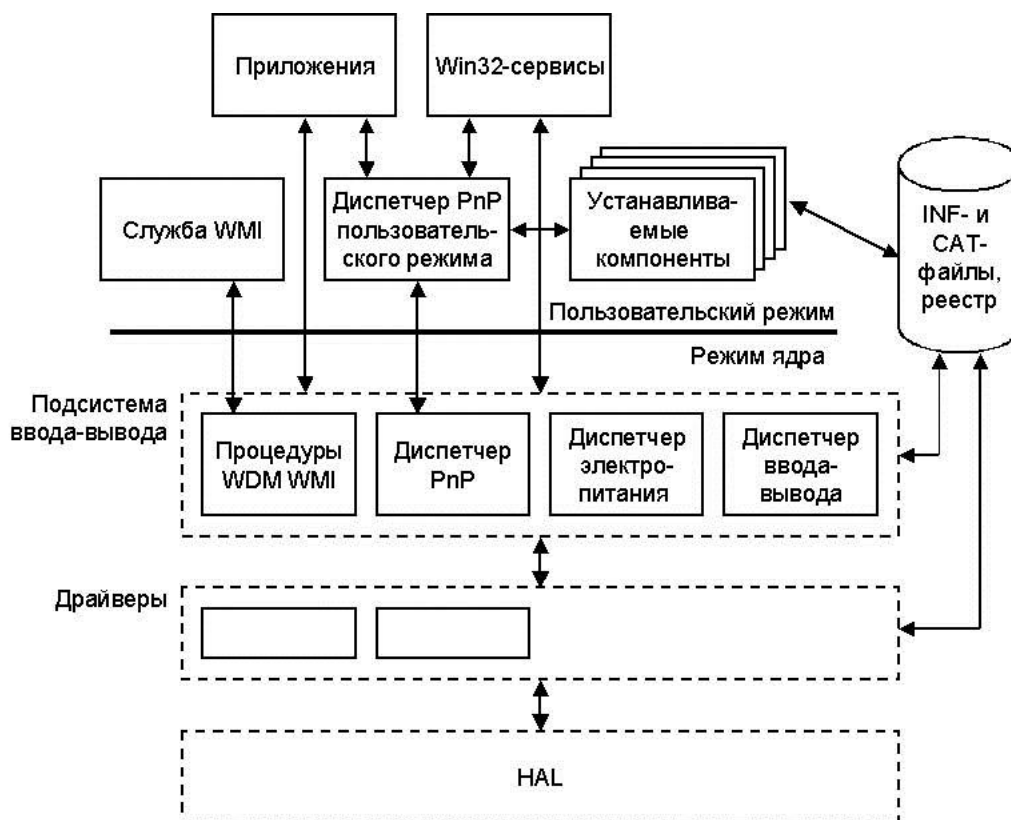


Рисунок 1.2 – Компоненты подсистемы ввода-вывода

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 Сервисы

Сервисы (Services), или службы, являются процессами, предоставляющими дополнительную функциональность в системе, не зависящую от интерактивных действий пользователя. То есть это приложения, запускаемые без учета того, зарегистрировался ли в системе какой-либо пользователь или нет, и довольно часто запуск самого сервиса происходит до момента появления окна регистрации пользователя. В части взаимодействия с системой сервисы используют «язык» Windows API и функционально состоят из трех компонентов:

- Service application — сервисное приложение (или драйвер);
- Service control program (SCP) — программа управления сервисом;
- Service Control Manager (SCM) — диспетчер управления сервисом.

Весь необходимый API для реализации механизма диспетчеризации (SCM-функции) сервисов сосредоточен в системной DLL-библиотеке Advapi32.dll (Advanced API).

Сервисное приложение — это драйвер или обычное Windows-приложение (чаще всего реализуемое как консольное), имеющее дополнительный блок кода, обеспечивающий обработку команд от SCP и возврат ему определенной статусной информации.

SCP — стандартное Windows-приложение, использующее для управления сервисом функции из набора SCM-функций. В основном SCP-приложение предназначено для запуска, останова и конфигурирования сервиса, иногда это приложение расширяет политику управления сервисом по отношению к реализуемой SCM-функции. В рамках операционной системы Windows существуют встроенные SCP, но для увеличения возможностей конфигурирования сервиса разработчик может написать свою собственную программу SCP, возможно, даже не реализуя ее как

отдельное приложение, а встраивая ее функции в основное приложение, использующее сервис. Сервисные приложения только косвенно взаимодействуют с SCP, изменяя свою статусную информацию в процессе выполнения команд от SCM. Перед использованием SCM-функций SCP необходимо установить канал связи с SCM (используется функция *OpenSCManager*). Важным моментом является то, что при установке любого приложения, в рамках которого предполагается использование сервиса, необходимо предварительно зарегистрировать сервис, вызвав функцию *CreateService* (реализована в *Advapi32.dll*). На основании параметров вызова этой функции SCM создает для каждого сервиса индивидуальный раздел в ветке реестра *HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\Service* (основная информационная база SCM, или ServicesActive database), куда помещается как параметр вновь созданного раздела реестра информация о месте расположения на диске исполняемого файла соответствующего сервиса, его параметры при старте и конфигурационные настройки (эти настройки помещаются в подраздел *Parameters* созданного для сервиса раздела).

SCM отвечает за загрузку как сервисов, так и драйверов, поэтому не удивительно, что значения некоторых параметров явным образом указывают на контекст текущего приложения. Общий термин (*Services*), используемый Microsoft для обозначения как служб, так и драйверов устройств (они действительно во многом схожи по стилю работы и характеру выполняемых функций), часто вызывает путаницу в понимании программной документации. Наиболее важные для нашей прикладной задачи параметры созданного раздела реестра и их возможные значения для драйверов и сервисов показаны в таблице 1.

Таблица 1.1 – Основные параметры при регистрации сервиса/драйвера и их возможные значения

Параметр	Значение	Описание
Start	SERVICE_BOOT_START (0)	Драйвер загружается Ntldr или Osloader, то есть, перед загрузкой ОС
	SERVICE_SYSTEM_START (1)	Драйвер загружается после загрузки и инициализации драйвера со значением
	SERVICE_AUTO_START (2)	Драйвер или сервис запускается SCM автоматически после запуска SCM-процесса (Strvices.exe)
	SERVICE_DEMAND_START (3)	Драйвер или сервис запускается SCM по требованию
	SERVICE_DISABLED (4)	Драйвер или сервис не загружается и не инициализируется
ErrorControl	SERVICE_ERROR_IGNORE (0)	Код ошибки, возвращаемый драйвером или сервисом, игнорируется
	SERVICE_ERROR_NORMAL (1)	Если драйвер или сервис возвращает код ошибки, выводится предупреждение
	SERVICE_ERROR_SEVERE (2)	Если драйвер или сервис возвращает и еще не использовалась последняя удачная конфигурация, то она используется. Если она уже используется, загрузка продолжается.
	SERVICE_ERROR_CRITICAL (3)	Если драйвер или сервис возвращает и еще не использовалась последняя удачная конфигурация, то она используется. Если она уже используется, выводится BSOD «Синий экран смерти»
Type	STRVICE_KERNEL_DRIVER (1)	Драйвер устройства режима ядра
	SERVICE_FILE_SYSTEM_DRIVER (2)	Драйвер файловой системы
ImagePath	Путь к исполняемому файлу	Путь к исполняемому файлу драйвера или сервиса. По умолчанию файл ищется в папке %SystemRoot\System32\Drivers
DisplayName	Имя сервиса	Имя сервиса. По умолчанию имя его раздела в реестре.

На конечном этапе загрузки операционной системы системный процесс Winlogon (перед появлением диалогового окна с приглашением к регистрации) запускает SCM, который в созданной внутренней базе данных сервисов SCM ищет записи драйверов и сервисов с параметром Start, имеющим значение SERVICE_AUTO_START, и запускает их.

1.3 Пример приложения, использующего сервис

В последующих двух разделах приведены примеры простейшего драйвера и работающей с ним программы-сервиса. Они осуществляют доступ к регистрам платы, установленной в слот ISA (например, модуля UNIOXX-5 фирмы Fastwel) [5]. Выбор устройства на шине ISA обусловлен более простой схемой программного взаимодействия с таким устройством. На самом деле, это может быть и плата, установленная в слот PCI, или любой другой системный ресурс, обращение к которому на уровне регистров запрещено из программ режима пользователя. При этом усложняется алгоритм взаимодействия с устройством, но принцип доступа через порты ввода-вывода остается. Драйвер написан на языке ассемблера, который наиболее эффективен при написании драйверов, где требования к минимизации кода и, как следствие, малое время исполнения объективно необходимы

На листинге 1.1 приведен пример реализации SCP на языке ассемблера. Минимально необходимый материал, позволяющий понять правила и нотацию для оформления программ на этом языке, можно найти в [3].

Листинг 1.1

; Пример простой программы управления сервисом (SCP)
.386

```
.model flat, stdcall
option casemap:none
include D:\masm32\INCLUDE\kernel32.inc
include D:\masm32\INCLUDE\user32.inc include
D:\masm32\INCLUDE\advapi32.inc
include D:\masm32\INCLUDE\windows.inc
includelib D:\masm32\LIB\kernel32.lib
includelib D:\masm32\LIB\user32.lib
includelib D:\masm32\LIB\advapi32.lib
.data
hSCManager          dd    0
hService             dd    0
ALIGN               4
CardUNIODriverPath  db    "D:\masm32\BIN\CardUNI0.sys",0
```



```

ALIGN                4
ErrMsg1              db  "Attempt of connection with SCM has
failed!",0
ALIGN                4
DrvName              db  "CardUNIO.sys",0
ALIGN                4
ErrMsg2              db  "Attempt to register the driver has failed!
",0
ALIGN                4
DrvName1             db  "CardUNIO",0
ALIGN                4
DispNameDrv          db  "MyDriver",0
.code
start                proc

; Устанавливаем канал связи с SCM
    invoke OpenSCManager, NULL, NULL,
SC_MANAGER_CREATE_SERVICE
    .if eax !=NULL
        mov hSCManager, eax
; Регистрируем драйвер
    invoke CreateService, hSCManager, offset DrvName1, \
offset DispNameDrv, \
SERVICE_START + DELETE, SERVICE_KERNEL_DRIVER, \
SERVICE_DEMAND_START, \
SERVICE_ERROR_IGNORE, addr CardUNIODriverPath, \
NULL, NULL, NULL, NULL, NULL
    .if eax != NULL
        mov hService, eax
; Запускаем драйвер
        invoke StartService, hService, 0, NULL
; Удаляем драйвер
        invoke DeleteService, hService
; Закрываем дескриптор драйвера
        invoke CloseServiceHandle, hService
    .else
; Если не удалось зарегистрировать сервис, выводим об этом
сообщение
        invoke MessageBox, NULL, offset ErrMsg2, NULL,
        MB_ICONSTOP
    .endif
; Закрываем канал связи с SCM
        invoke CloseServiceHandle, hSCManager
    .else
; Сообщение в случае невозможности установить связь с SCM
    invoke MessageBox, NULL, offset ErrMsg1, NULL, MB_ICONSTOP
    .endif
        invoke ExitProcess, 0
start endp
end start

```

Действия программы заключаются в следующем. Устанавливается канал связи с SCM. Если при установлении канала связи происходит ошибка, выводится сообщение “Attempt of connection with SCM has failed!” и программа завершается. В случае установления канала регистрируем драйвер. Если происходит ошибка регистрации, выводим соответствующее сообщение “Attempt to register the driver has failed!” и, закрыв дескриптор SCM, завершаем программу. В случае успеха регистрации драйвера запускаем драйвер, удаляем его, закрываем дескриптор драйвера, закрываем дескриптор SCM, завершаем программу. В этой программе драйвер запускается один раз, и поэтому его действия носят характер, только подтверждающий факт его работы. Теперь более подробно.

Для вызова SCM-функций SCP должна установить канал связи с SCM, используя функцию *OpenSCManager*.

Прототип функции выглядит следующим образом:

OpenSCManager proto

lpMachineName:LPSTR,

lpDatabaseName:LPSTR,

dwDesiredAccess:DWORD

lpMachineName

Указатель на строку, завершающуюся нулем, содержащую имя компьютера. Устанавливая этот параметр в NULL, мы устанавливаем связь с локальным SCM (на этом компьютере).

lpDatabaseName

Указатель на строку, завершающуюся нулем, содержащую имя открываемой базы. Устанавливая этот параметр в NULL, мы устанавливаем связь с локальной, активной в текущий момент базой данных – SERVICES_ACTIVE_DATABASE.

dwDesiredAccess

Права доступа, запрашиваемые при открытии канала. Могут быть следующие значения:

SC_MANAGER_CONNECT (устанавливаются по умолчанию, параметр 0);

SC_MANAGER_CREATE_SERVICE (доступ для внесения в базу данных записи о новом драйвере);

SC_MANAGER_ALL_ACCESS (полный доступ).

Если эта функция возвращает не NULL (NULL говорит об ошибке), то мы получаем дескриптор активной базы данных. Следующий шаг — это регистрация нашего драйвера путем вызова функции CreateService, прототип которой выглядит так:

```
CreateService proto  
hSCManager:HANDLE,  
lpServiceName:LPSTR,  
lpDisplayName:LPSTR,  
dwDesiredAcces:DWORD,  
dwServiceType:DWORD,  
dwStartType:DWORD,  
dwErrorControl:DWORD,  
lpBinaryPathName:LPSTR,  
lpLoadOrderGroup:LPSTR,  
lpdwTagId:LPDWORD,  
lpDependencies:LPSTR,  
lpServiceStartName:LPSTR,  
lpPassword:LPSTR  
hSCManager  
Дескриптор базы данных SCM.  
lpServiceName
```

Указатель на строку, завершающуюся нулем, содержащую имя драйвера/сервиса. Длина до 256 символов. Соответствует имени подраздела в реестре.

lpDisplayName

Указатель на строку, завершающуюся нулем, содержащую экранное имя драйвера/сервиса. Длина до 256 символов. Соответствует значению параметра DisplayName в реестре.

dwDesiredAcces

Запрашиваемый тип доступа. Могут быть значения:

SERVICE_ALL_ACCESS (полный доступ);

SERVICE_START (доступ на запуск драйвера/сервиса);

SERVICE_STOP (доступ на останов драйвера/сервиса);

DELETE (доступ на удаление драйвера/сервиса из базы SCM).

dwServiceType

Тип сервиса, в нашем случае SERVICE_KERNEL_DRIVER.

Соответствует значению параметра TYPE в реестре.

dwStartType

Тип запуска, в нашем случае SERVICE_DEMAND_START.

Соответствует значению параметра START в реестре.

dwErrorControl

Характер контроля ошибок, в нашем случае SERVICE_ERROR_IGNORE. Соответствует значению параметра ErrorControl в реестре.

lpBinaryPathName

Указатель на строку, завершающуюся нулем, содержащую полный путь к файлу загружаемого драйвера. Соответствует значению параметра ImagePath в реестре.

lpLoadOrderGroup

Указатель на строку, завершающуюся нулем, содержащую имя группы, в случае если загружаемый драйвер является членом группы. В противном случае значение NULL или указатель на пустую строку.

lpdwTagId

Указатель на переменную, содержащую уникальное значение тега, идентифицирующее группу. В противном случае NULL.

lpDependencies

Указатель на массив имен драйверов или групп драйверов, загрузка которых должна быть осуществлена до запуска текущего драйвера. Массив заканчивается двумя нулями, имена драйверов или групп разделяются одним нулем. Если запуск драйвера не связан с предварительным запуском других драйверов, значение NULL.

lpServiceStartName

Указатель на строку, завершающуюся нулем, содержащую имя учетной записи (account), с правами которой запускается текущий драйвер. В случае типа сервиса SERVICE_KERNEL_DRIVER параметр содержит имя объекта драйвера. Если используется имя объекта драйвера, присвоенное подсистемой ввода-вывода, то NULL.

lpPassword

Указатель на строку, завершающуюся нулем, содержащую пароль учетной записи, с правами которой запускается текущий драйвер. В случае SERVICE_KERNEL_DRIVER значение этого параметра игнорируется.

Остальные функции — *StartService*, *DeleteService* и *CloseServiceHandle* — запускают, удаляют и закрывают дескриптор нашего

драйвера. Далее приводятся прототипы функций и описания их параметров.

StartService proto

hService:HANDLE,

dwNumServiceArgs:DWORD,

lpServiceArgVectors:LPSTR

hService

Дескриптор драйвера/сервиса.

dwNumServiceArgs

Количество аргументов, передаваемых сервису. Драйверу не передаются аргументы, поэтому значения NULL.

lpServiceArgVectors

Указатель на массив указателей, ссылающихся на строки, завершающиеся нулем. В строках содержатся передаваемые службе аргументы. В нашем случае аргументы отсутствуют, поэтому значение NULL.

DeleteService proto

hService:HANDLE

hService

Дескриптор удаляемой службы.

CloseServiceHandle proto

hSCObject:HANDLE

hSCObject

Дескриптор закрываемого SCM, сервиса, драйвера.

1.4 Прототип драйвера режима ядра

Рассмотрим шаблон простейшего драйвера (листинг 1.2) [4].

Листинг 1.2

```
; CardUNIO.asm
.386
.model flat, stdcall
option casemap:none
include D:\masm32\INCLUDE\DDK\wxp\ntddk.inc

UNIO_FPGA1_BaseAddr equ 0a110h
; базовый адрес FPGA1 UNIOxx-5 при поставке
MyDelay equ 100h

STATUS_IO_DEVICE_ERROR equ 0C0000185h

.code
ShortDelay proc ValDelay:DWORD
    push cx
    mov cx, ValDelay
@@delay:
    dec cx
    jnz @@delay
    pop cx
    ret
ShortDelay endp
EntryDriverUNIO proc DriverObject:PDRIVER,OBJECT, \
RegistryPath: PUNICODEJTRING
    push cx
    push dx
    xor al,al
    inc al
    mov dx, UNIO_FPGA1_BaseAddr
    ; в контрольном регистре устанавливаем BANK = 1
    cli
    out dx,al
    sti
    invoke ShortDelay, MyDelay
    ; в регистрах маски каналы 0-23 на вывод
    inc dx
    mov al,0ffh
    cli
    out dx,al
    sti
    invoke ShortDelay, MyDelay
    xor al,al
    dec dx
    ; в контрольном регистре устанавливаем BANK = 0
    cli
    out dx,al
```

```

    sti
    invoke ShortDelay, MyDelay
    inc dx
    ; формируем импульс на 0-7 выходах
    cli
    out dx.al
    invoke ShortDelay, MyDelay
    mov al.Offh
    out dx.al
    invoke ShortDelay, MyDelay
    xor al.al
    out dx.al
    ; возвращаем системе сообщение о неудачной инициализации
драйвера
    mov eax, STATUS_IO_DEVICEJRROR
    pop dx
    pop cx
    ret
EntryDriverliNIO endp
End EntryDriverUNIO

```

Если читатель знаком с динамически подключаемой библиотекой (DLL) [3], то приведенный пример во многом напомнит структуру простейшей динамической библиотеки. Надо сказать, что между драйверами и DLL очень много общего. После загрузки драйвера операционная система передает управление на его точку входа. Предполагается, что выполняемая функция программы, которой передано управление сразу после загрузки драйвера, — это инициализация структур и переменных, необходимых для дальнейшей работы драйвера, и отчета о выполненной задаче (“DriverEntry is the first routine called after a driver is loaded, and is responsible for initializing the driver”, — читаем мы в MSDN). Точкой входа является метка, указанная после директивы End, то есть в нашем случае это EntryDriverUNIO (может быть и другое имя). В приведенном примере после передачи управления на точку входа происходит формирование перехода из 0 в 1 на одном из выходов FPGA1-платы UNIOXX-5 (используется стандартная прошивка g00). Реальность отработки драйвера можно наблюдать, подключив осциллограф или светодиод к соответствующему выходу платы. Естественно, для

визуализации работы драйвера можно было обойтись и без платы, например, вывести какое-либо сообщение, или озвучить это событие на динамике ПК. Затем в регистр `eax` загружается возвращаемый параметр (`STATUS_IO_DEVICE_ERROR`), тем самым системе сообщают об ошибке. На этом работа (и «жизнь») драйвера завершается. Так как текущий пример является только шаблоном драйвера, каких-либо конструктивных действий в драйвере не производится. Код ошибки выбран произвольно, его использование имело целью сообщить системе о том, что работа драйвера не может быть продолжена и его необходимо выгрузить, что операционная система и сделает. Но главное достигнуто. Совершенно корректно, используя все «джентльменские» правила поведения в операционной системе Windows, получен полный доступ в святая святых — к системным ресурсам операционной системы, имея нулевой уровень привилегий для исполняемого кода. Бездумное использование открывшихся возможностей может привести к неожиданным результатам, в том числе, и к краху системы.

Можно свободно читать и записывать любые данные из внешних портов операторами `IN` и `OUT`, что было так привычно в DOS, записывать данные в любые адреса памяти, но одним неподходящим оператором или некорректным значением, записанным в порт или операцию память, можно получить «синий экран смерти» (BSOD) на мониторе. Большие возможности предполагают и большую ответственность.

Рассмотрим прототип `DriverEntry` (у нас это процедура `EntryDriverUNIO`).

DriverEntry proto

DriverObject:PDRIVER_OBJECT,

RegistryPath:PUNICODE_STRING

DriverObject

Указатель на объект созданного драйвера. Если говорить проще, то речь идет о структуре типа DRIVER_OBJECT, описанной в файле NTDDK.h DDK. Часть полей в этой структуре необходимо заполнить загруженному драйверу, именно по этой причине ему и передается указатель на эту структуру. В нашем примере мы не занимались этой работой, так как «не задерживаемся надолго». В полнофункциональном драйвере эту работу придется проделать.

RegistryPath

Указатель на структуру типа UNICODE_STRING, содержащую указатель на UNICODE-строку, в которой содержится имя раздела в реестре с параметрами инициализации драйвера.

Надо особо подчеркнуть, что, в отличие от пользовательского режима, в режиме ядра операционная система работает только со строками типа UNICODE_STRING. Теперь несколько слов о компиляции и компоновке драйвера. Строка компиляции достаточно традиционна и, если мы работаем с masm32, выглядит следующим образом:

```
ML /nologo /c /coff CardUNIO.asm
```

Опции компоновщика, естественно, отличаются от опций для стандартного исполняемого файла:

```
LINK /nologo /driver /base:0x1000  
/out:CardUNIO.sys /subsystem:native  
CardUNIO.obj  
/driver
```

Выходной файл – драйвер.

/base:0x1000

Предопределенный адрес загрузки драйвера.

/out:CardUNIO.sys

Выходной файл драйвера должен иметь соответствующее расширение (по умолчанию — .exe).

/subsystem:native

Тип подсистемы, необходимый для работы выходного файла. Этот вопрос был кратко рассмотрен в разделе, посвященном обзору архитектуры Windows. При этом речь шла о существовании трех подсистем окружения: Win32, POSIX и OS/2. Параметр Native говорит о том, что нет необходимости ни в одной из этих подсистем. Драйвер работает в «родной» или естественной среде, то есть использует базовый API самой операционной системы Windows.

1.5 Контрольные вопросы к разделу

1. Что такое привилегированный режим?
2. Что такое пользовательский режим?
3. Каков диапазон адресов, используемых приложениями?
4. Возможен ли доступ к ресурсам системы из пользовательского режима?
5. Возможен ли доступ к оборудованию из пользовательского режима?
6. Вся ли операционная система Windows NT5 работает в привилегированном режиме?
7. В каком режиме работают системы поддержки?
8. В каком режиме работают сервисы системы?
9. В каком режиме работают подсистемы окружения?
10. Могут ли драйверы работать в пользовательском режиме?
11. Какие компоненты системы работают в режиме ядра?
12. Для чего нужен уровень аппаратных абстракций?
13. Какие категории драйверов режима ядра имеются в Windows?
14. Для чего нужна поддержка PnP?
15. Для чего нужна поддержка ACPI?
16. Что такое Windows Driver Model?
17. Какие типы драйверов предусматривает Windows Driver Model?
18. Что такое драйвер шины?
19. Что такое функциональный драйвер?
20. Для чего нужен драйвер фильтра?
21. Для чего нужны inf-файлы?
22. Для чего нужны cat-файлы?
23. Из каких компонентов состоит подсистема ввода-вывода Windows?
24. Что такое сервисы?
25. Что такое сервисное приложение?
26. Для чего нужна программа управления сервисом SCP?
27. Для чего нужен диспетчер управления сервисом SCM?

- 28.Какие параметры используются при регистрации сервиса или драйвера?
- 29.Какие значения может иметь параметр Start?
- 30.Какие значения может иметь параметр ErrorControl?
- 31.Какие значения может иметь параметр Type?
- 32.Какие функции выполняет рассмотренный пример SCP?
- 33.Как устанавливается канал связи с SCM?
- 34.Что происходит в случае установления канала связи?
- 35.Что происходит в случае неустановления канала связи?
- 36.Что происходит, если не удастся зарегистрировать драйвер?
- 37.Какова структура прототипа драйвера ядра?
- 38.Что делает операционная система после загрузки драйвера?
- 39.Что такое точка входа драйвера?

2 ИСПОЛЬЗОВАНИЕ ПАКЕТА NUMEGA DRIVER STUDIO ДЛЯ НАПИСАНИЯ WDM- ДРАЙВЕРОВ УСТРОЙСТВ

Разработка WDM – драйвера с использованием только DDK является сложной и трудоемкой задачей. При этом приходится выполнять много однотипных операций: создание скелета драйвера, написание inf – файла для его установки, создание приложения для тестирования и т.п. При этом многие из этих операций однотипны и стандартны. Часто при написании драйверов приходится выполнять однотипные операции. Например, если разрабатывается драйвер устройства для шины PCI, то наверняка придется сделать:

- вручную написать .inf-файл для инсталляции драйвера;
- выполнить конфигурацию устройства при запуске драйвера: выполнить проверку, присутствуют ли необходимые ресурсы (память, порты, запросы на IRQ в устройстве);
- написать процедуры управления энергопотреблением (если они нужны);
- прочитать из реестра Windows необходимую конфигурацию;
- написать программу для тестирования работоспособности драйвера (хотя бы для проверки, правильно ли он проинсталлирован и правильно ли обрабатывает основные запросы).

Все перечисленные операции – рутинная, однотипная работа, стандартная для большинства драйверов.

Для ускорения проектирования и разработки драйверов устройств под Windows используются программные пакеты разных фирм. Наиболее известным пакетом является программа DriverStudio фирмы NuMega. Для работы этой программы обязательной является установка пакета DDK

(желательно – DDK 2000 как наиболее универсального) и среды Visual C++ версии не ниже 5.0. Желательно использовать такую конфигурацию [6]:

- DriverStudio 2.01(далее в тексте – DS);
- DDK 2000;
- Visual C++ 6.0 (далее в тексте – VC++).

В нее входят следующие программы:

DriverWorks – эта программа является основным компонентом DriverStudio. Именно с помощью DriverWorks выполняется разработка драйвера под Windows 98/ME/2K с использованием WDM. Установка этой программы обязательна. При инсталляции DriverWorks интегрируется в среду разработки Visual C++.

VtoolsD – средство для разработки .vxd – драйверов. Данная утилита не зависит от других программ DS или VC++ и может не устанавливаться. В принципе, если не предполагается разработка .vxd – драйверов, данный компонент не является необходимым.

SoftIce – kernel-mode отладчик. Эта программа может быть использована как для отладки драйверов, так и в других целях. Фактически это очень мощный отладчик, который может получать доступ к практически любым элементам системы. Недостатками его является его высокая сложность и неудобство в эксплуатации. Работа с SoftIce бывает опасна именно в силу его больших возможностей: любое неверное действие обычно фатально для системы. Хотя, для отладки драйверов устройств трудно найти что-либо лучшее.

DriverNetworks – используется для разработки драйверов сетевых устройств. Если не предполагается разработка такого драйвера, данный компонент не является необходимым.

Для инсталляции DS на компьютере необходимо проинсталлировать пакет DDK и среду VC++. После этого можно начинать инсталляцию DS. Сама инсталляция проста и не отличается от процесса инсталляции того же

VC++, например. По умолчанию пакет ставится в папку C:\Program Files\NuMega\DriverStudio. Знание пути к DS необходимо для дальнейшей работы с программой. В дальнейшем мы будем его называть *<путь_к_DS>*. При первом запуске DS необходимо скомпилировать библиотеки, необходимые для работы. Для этого следует запустить среду VC++ и открыть проект *<путь_к_DS>\DriverVorks\source\vdwlibs.dsw*. Вся суть в том, что DS использует собственную библиотеку классов для написания драйвера. Эта библиотека поставляется в исходных кодах, подобно библиотеке MFC или библиотекам под UNIX. Поэтому теперь необходимо откомпилировать с помощью VC++ данный проект.

Стоит сразу проверить опции проекта и установить активную конфигурацию VdwLibs – Win32 WDM Checked (если планируется отлаживать скомпилированные драйвера) или Win32 WDM Free. Теперь запускаем проект на компиляцию. В результате в папке *<путь_к_DS>\DriverVorks\lib\i386\checked* появляется библиотека *vdw.lib* (при использовании ОС win2K) или *vdw_wdm.lib* (win 9x). DS готов к работе.

2.1 Система классов DriverWorks

Возможно, идея писать объектно-ориентированный драйвер и кажется на первый взгляд нелогичной, однако, при более близком знакомстве с DriverStudio и с драйверами в целом, оказывается, что это не так уж страшно и довольно удобно. Объектная модель DriverWorks отражает архитектуру WDM и представляет собой систему классов, построенную на системных вызовах. Цель DriverWorks, с одной стороны, оставаться на достаточно низком уровне программирования, чтобы эффективно писать драйверы, а с другой – упростить и упорядочить процесс разработки драйверов режима ядра.

В соответствии с идеологией DriverWorks драйвер представляется, как набор объектов. Эта же идея присутствует и в "чистой" архитектуре WDM, но DriverWorks упорядочивает эти объекты и представляет их экземплярами классов. Классы DriverWorks также несколько упрощают код драйвера по сравнению с DDK, делают его более компактным и доступным для понимания. Часто повторяющиеся, рутинные фрагменты кода драйвера спрятаны внутри методов класса. И то, что при использовании пакета DDK занимало несколько строк в программе, теперь можно вполне заменить вызовом одного единственного метода.

Также в DriverWorks предложено несколько полезных классов: например класс KFile – доступ к файлам или классы динамических списков и массивов.

Сама идея DriverWorks напоминает Visual C++ и библиотеку MFC. MFC представляет собой некую прослойку, которая отделяет программиста от жутковатых функций API и позволяет создавать объектно-ориентированные проекты, при этом оставаясь на достаточно низком уровне программирования.

Впрочем, в системе классов DriverWorks есть одна особенность – иерархия классов практически отсутствует. Это вполне естественно: в

системе классов DriverWorks присутствуют самые различные классы – классы, представляющие собой ресурсы устройства (линии ПДП, прерываний, областей памяти, портов ввода-вывода), сами устройства, классы для взаимодействия с реестром, файлами и т.п. Еще одним аргументом в пользу отсутствия наследования является то, что разветвленная иерархия классов может снизить быстродействие программы. Для драйвера, это, конечно, неприемлемо.

В основе архитектуры DriverWorks лежит несколько основных классов.

Объект драйвера (Driver Object)

Объект драйвера является экземпляром класса KDriver. Он представляет драйвер в целом как некую абстракцию. Для объекта драйвера абсолютно все равно, каким оборудованием он управляет, объект драйвера об этом по-настоящему никогда не задумывается. Его задача – обеспечить интерфейс драйвера с операционной системой: загрузку и инициализацию драйвера, его выгрузку и т.п. А управление аппаратурой возлагается на другие объекты драйвера, в частности, на объект устройства.

Когда операционная система загружает драйвер, то она создает для него соответствующий объект драйвера [5]. Компонент ядра операционной системы – диспетчер ввода-вывода (I/O Manager) – использует объекты драйверов для управления устройствами. Каждый драйвер отвечает за управление одним или несколькими объектами устройств. Запрос на операцию ввода-вывода (I/O request), посланный приложением пользователя, поступает к диспетчеру ввода-вывода.

Диспетчер ввода-вывода определяет, какой именно объект драйвера отвечает за соответствующий объект устройства, и перенаправляет ему запрос. Кроме управления объектами устройств, объект драйвера имеет дополнительные методы, отвечающие за инициализацию и завершение

работы драйвера. Программист создает свой подкласс класса KDriver для взаимодействия с системой. Он обязательно должен содержать метод DriverEntry – функцию, вызываемую при инициализации драйвера.

В отличие от обычного не-WDM драйвера, процедура инициализации WDM-драйвера выполняет весьма ограниченное число функций: в основном это загрузка некоторых внутренних переменных на основе данных реестра. WDM-драйвер не инициализирует ресурсы устройства при старте в вызове EntryPoint. Для этого существует объект устройства.

WDM-драйвер экспортирует метод AddDevice, который вызывается системой, если обнаружено устройство, поддерживаемое данным драйвером. Этот метод отвечает за создание объектов устройств, соответствующих системным физическим объектам устройств (Physical Device Object, PDO).

Объект драйвера может содержать дополнительные методы для реинициализации драйвера на более поздних стадиях загрузки системы. Такой подход необходим, если в системе присутствует несколько драйверов, критичных к порядку их загрузки. Впрочем, такие проблемы встречаются нечасто.

В принципе, хорошо спроектированный драйвер должен экспортировать метод Unload, который вызывается при выгрузке драйвера. Но такие случаи встречаются довольно редко.

Класс KRegistryKey

Как уже было сказано, драйвер обращается к системному реестру при инициализации. Системный реестр (registry) – системная база данных, организованная в виде дерева, похожего на дерево каталогов. Каждую ветвь этого дерева (реестра) называют разделом (key), каждый лист – параметром (value). Данные, хранящиеся в реестре, могут быть разных типов: целое (integer), строка, набор байтов.

Система позволяет каждому драйверу хранить данные в реестре. Эти данные используются драйверами при старте и инициализации. Обычно драйвер хранит данные в разделе HKLM\SYSTEM\CurrentControlSet\Services\<имя драйвера>\Parameters\.

В DriverWorks есть класс KRegistryKey, который облегчает доступ к параметрам реестра. Он имеет методы для чтения (QueryValue), записи (WriteValue), удаления (Delete) значений ключей реестра. При вызове конструктора KRegistryKey сразу указывается ключ, с которым будет связан создаваемый объект. Далее можно изменить ключ при помощи метода Reconstruct.

Объект запроса на ввод-вывод (I/O Request Object)

Объекты запроса на ввод-вывод, более известные, как пакеты запроса на ввод-вывод (I/O request packet, IRP – так мы и будем их называть в дальнейшем), предназначены для управления драйверами режима ядра.

Физически IRP представляет собой весьма сложную структуру данных, содержащую множество полей, таких как код статуса, указатель на буфер пользователя, указатель на IRP драйвера более высокого уровня, различные флаги и т.п. Многие из этих полей не используются драйверами режима ядра, но необходимы для того, чтобы IRP был функционально полным инструментом управления драйверами. То есть, при помощи IRP можно управлять любым типом драйвера.

Обмен информацией и управление драйверами при помощи IRP выглядит примерно следующим образом: когда приложение пользователя посылает данные или пытается получить данные из устройства, диспетчер ввода-вывода формирует IRP и отправляет его драйверу, отвечающему за данное устройство. Объект драйвера получает этот IRP и перенаправляет его одному из своих объектов устройств. Объект устройства, получив пакет, может либо начать его обработку немедленно, либо поставить его в

очередь, чтобы обработать этот пакет позже. Что именно сделает объект устройства, зависит от того, какой пришел IRP, от состояния объекта устройства и от состояния самого устройства. После того, как пакет будет обработан, объект устройства пошлет IRP с информацией о результате операции обратно диспетчеру ввода-вывода.

Каждый IRP описывает операцию ввода-вывода, которая может быть выполнена устройством. Для того, чтобы драйвер смог получить информацию о том, какая именно операция должна быть выполнена, IRP содержит целый набор атрибутов: старший и младший коды функции, код статуса и различные параметры: число байтов, которые должны быть прочитаны, смещение и т.п. За время своего существования IRP может проходить несколько уровней иерархии драйверов устройств в системе. Поэтому в пакете резервируется место для сохранения данных и параметров, необходимых для следующего драйвера в иерархии – так называемый "стек IRP", "IRP stack location". Когда объект устройства обрабатывает запрос, то он имеет доступ только к тем участкам стека, которые предназначены для использования им или устройством более низкого уровня, которому будет перенаправлен IRP.

IRP могут создаваться как диспетчером В/В, так и самими драйверами. Чаще всего это происходит при выполнении функций CreateFile, CloseFile, ReadFile, WriteFile и DeviceControl.

IRP может быть уничтожен, если необходимо отменить операцию ввода-вывода, например, при закрытии приложения. Объект IRP содержит указатель на функцию, вызываемую при уничтожении пакета.

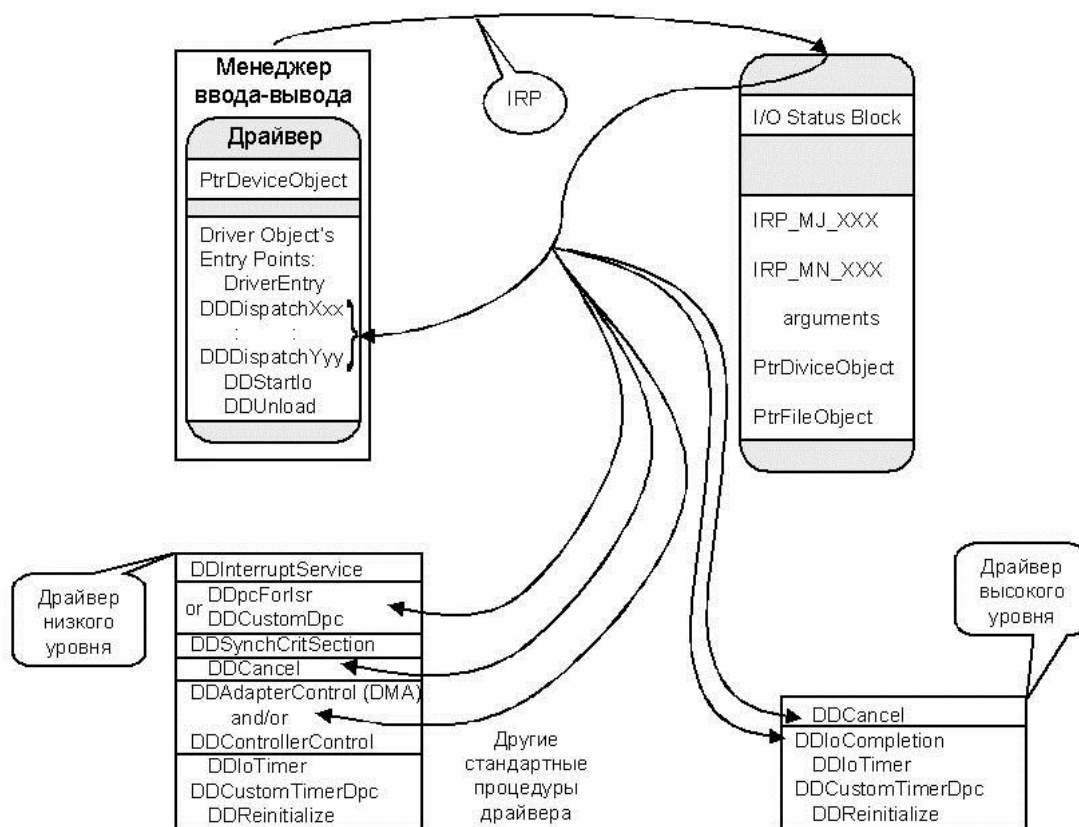


Рисунок 2.1 – Интерфейс с драйвером при помощи IRP

Объект устройства (Device Object)

Объекты устройств являются экземплярами класса KDevice или KPnpDevice. Эти классы являются краеугольными камнями архитектуры DriverWorks: они представляют собой как бы программный образ тех устройств, которые присутствуют в системе. Именно объекты устройств обеспечивают управление и обмен данными с внешними устройствами, управление их ресурсами – линиями прерываний, каналами ПДП, диапазонами адресов памяти, портами ввода-вывода и т.п. Когда выполняется системный вызов типа CreateFile, ReadFile, WriteFile, диспетчер ввода-вывода посылает IRP соответствующему драйверу. Но сам драйвер, вернее объект драйвера, не выполняет никаких операций по обработке этого пакета – он просто передает его объекту устройства и забывает о самом существовании этого IRP. Это естественно, ведь

управление физическим устройством – не его задача, это дело соответствующего объекта устройства.

Класс KDevice является суперклассом для всех классов устройств. Но на практике он сейчас почти не используется. Чаще используют его потомка – класс KPnpDevice. Этот класс предназначен для управления PnP-устройствами, т.е. устройствами, которые конфигурируется системой. В данный момент практически все устройства являются PnP-устройствами. Появление таких устройств сильно облегчило жизнь разработчикам драйверов: использовать KPnpDevice намного проще, а часто и безопаснее, чем KDevice. В данном случае все проблемы конфигурирования и инициализации ресурсов оборудования ложатся на плечи системы.

Любой объект устройства содержит стандартные методы обработки запросов на чтение, запись и управление устройством (device control). Эти методы вызываются при вызове соответствующих функций API ReadFile(), WriteFile(), DeviceControl().

Каждый драйвер содержит минимум один объект устройства. Как уже говорилось, объект устройства является экземпляром класса, порожденного программистом от класса KDevice или KPnpDevice. Для придания функциональности объекту устройства разработчик драйвера может переопределять виртуальные методы суперкласса, включая те методы, которые обрабатывают различные типы IRP, а также добавлять свои свойства и методы в класс. В процессе разработки можно использовать как иерархию классов DriverStudio, так и функции DDK. Впрочем, для большинства задач без использования DDK вполне можно обойтись.

Естественно, делать все это надо с осторожностью. Вызов некоторых методов является обязательным. Переопределение виртуальных методов также требует осторожности: многие из них содержат код, который обязательно должен быть выполнен. Если его удалить, то драйвер будет

работать неправильно (или не будет работать вообще). В результате, система, скорее всего, зависнет.

Остается открытым вопрос, как же все-таки драйвер, вернее объект устройства, управляет аппаратурой? Официально провозглашается, что Win2000 – переносимая система. То есть, если она хорошо работает на архитектуре Intel, то она также может быть перенесена и на другие системы, например Alpha. Для того, чтобы система с минимальными исправлениями могла работать на компьютерах с другой архитектурой, и был введен HAL – уровень абстракции аппаратуры. Он действительно абстрагирует драйвера и большую часть кода ядра операционной системы от того, как именно построен компьютер. Теперь разработчику драйвера становится абсолютно все равно, как на данном компьютере реализован контроллер прерываний или контроллер прямого доступа к памяти – все ресурсы аппаратуры также представлены объектами. Это диапазоны адресов памяти и портов ввода-вывода устройств, линии прерываний и прямого доступа к памяти (ПДП). Все они могут быть реализованы в различных архитектурах по-разному, но общие принципы их работы остаются одними и теми же. Соответственно, и интерфейс классов, реализующих управление этими ресурсами, остается одинаковым. Программистам, теперь не нужно знать тонкости работы аппаратной части на этом компьютере – это задача HAL и тех людей, которые переносили систему.

На практике это означает, что если устройство, например, имеет диапазон адресов памяти и линию запроса на прерывание, то класс устройства будет содержать два свойства (данные). Одно из них – экземпляр класса KMemoryRange, который будет реализовывать управление памятью устройства, а другое – экземпляр класса KInterrupt, который управляет линией запроса на прерывание, и всем, что с ней связано. Если устройство будет иметь несколько областей адресов памяти,

то, соответственно, класс устройства будет содержать несколько экземпляров класса KMemoryRange.

Другим способом управления устройствами является наличие устройств нижнего уровня (Lower devices). Как уже было отмечено, особенностью архитектуры WDM является наличие стека драйверов, когда драйверы могут обмениваться IRP-пакетами между собой. Данную ситуацию легче объяснить при помощи рисунка:

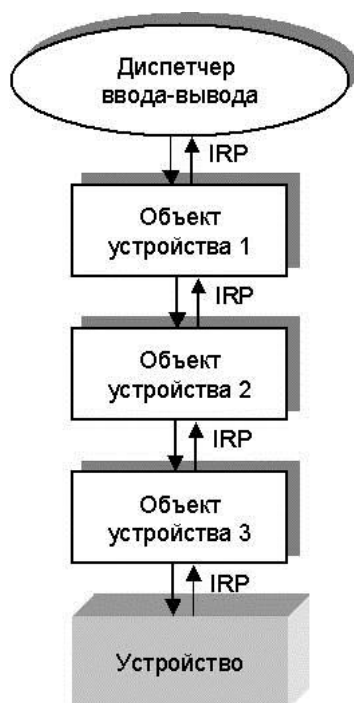


Рисунок 2.2 – Стек устройств

На рисунке 2.2 изображен стек устройств, состоящий из трех объектов устройств. Устройство 1 – самое первое (верхнее) в стеке, устройство 3 – самое последнее (нижнее) в стеке. Тогда по отношению к устройству 1 устройство 2 будет устройством нижнего уровня. Устройства верхнего уровня для устройства 1 нет. Устройство 2 имеет и устройство верхнего уровня (устройство 1) и устройство нижнего уровня (устройство 3). Для устройства 3 есть только устройство верхнего уровня (устройство 2), устройства нижнего уровня у него нет, устройство 3 напрямую контролирует оборудование.

Такой метод управления оборудованием, когда в системе присутствует не один драйвер, а целая цепочка драйверов, может иметь свои преимущества. Предположим, наше физическое устройство – это клавиатура, подключенная к USB – порту. Тогда объект устройства 3 – драйвер USB – порта. Устройство 2 выполняет действия, специфичные именно для данного типа клавиатур: читает данные из портов ввода-вывода клавиатуры, "висит" на прерывании, выполняет дополнительные функции (например, если у нас мультимедийная клавиатура). Он передает коды нажатых клавиш устройству 1. Устройство 1 не зависит от того, какой именно тип клавиатуры подключен к компьютеру. Оно реализует очередь кодов нажатых клавиш; реакцию на клавиши CapsLock, Shift и т.п.

Если в данном случае у компьютера поменяется клавиатура, то необходимо установить только новый драйвер 2. Если клавиатура переключится на другой порт, то устройство 2 будет общаться не с устройством 3, а с каким-то другим устройством. В таком случае система становится более гибкой, легкой в проектировании, более надежной и простой в использовании. И пользователю, и приложениям становится абсолютно все равно, какой тип клавиатуры установлен на компьютере.

В нашем примере получается, что и устройство 1, и устройство 2 управляет оборудованием – клавиатурой. Но они делают это не напрямую, а посылая IRP устройству 3. Для того, чтобы наш объект устройства мог передавать IRP- пакеты другим объектам устройств, введен класс устройств нижнего уровня (KLowerDevice, KPnpLowerDevice). Естественно, для этого устройство должно знать, как управлять устройством нижнего уровня при помощи IRP.

Впрочем, подобная ситуация имеет место практически во всех современных операционных системах. Только в других системах это выражено менее ярко и не декларируется, как "официальная идеология".

Затрагивая тему управления аппаратурой, нельзя не упомянуть еще об одном способе управления устройствами. Иногда нет возможности использовать классы DriverWorks или функции DDK. Например, необходимо обратиться непосредственно к портам ввода-вывода компьютера, в частности, к портам управления принтером. Напрямую сделать это из приложения пользователя, работающего под Winodws NT5, невозможно. Все пользовательские программы работают в непривилегированном кольце защиты 3 и не могут выполнять ассемблерные команды типа inp/outp. Но драйвер работает в кольце защиты 0 и, фактически, может делать все что угодно. В этом случае следует переопределить методы класса устройства, например ReadFile(), WriteFile(), DeviceControl() – добавить туда ассемблерные вставки или код на Си, выполняющий то, что нам необходимо сделать (чаще всего это обращение к портам ввода-вывода). Впрочем, любое обращение к портам ввода-вывода компьютера напрямую может оказаться опасным. Если программист допустит ошибку или неточность в манипуляциях с параллельным портом, то это, скорее всего, пройдет бесследно для системы. Но если он ошибется при обращении к портам управления таймером, винчестером или другими жизненно важными устройствами компьютера, то в лучшем случае система зависнет.

Объекты очередей и буферизация запросов

Сколько операций может параллельно выполнять наше физическое устройство? Естественно, это определяется самой природой этого устройства. Многие виды оборудования могут одновременно делать что-то одно. Например, параллельный порт не может передать два байта за один раз при всем нашем желании, ведь физически это один канал передачи. Но ведь IRP – пакеты могут приходить в любое время! Поэтому большинство объектов устройств должны содержать какой-либо механизм для буферизации и упорядочивания (serialization) запросов, так как зачастую

только один запрос может быть обработан в единицу времени. Самым простым и в то же время эффективным методом такой буферизации является очередь.

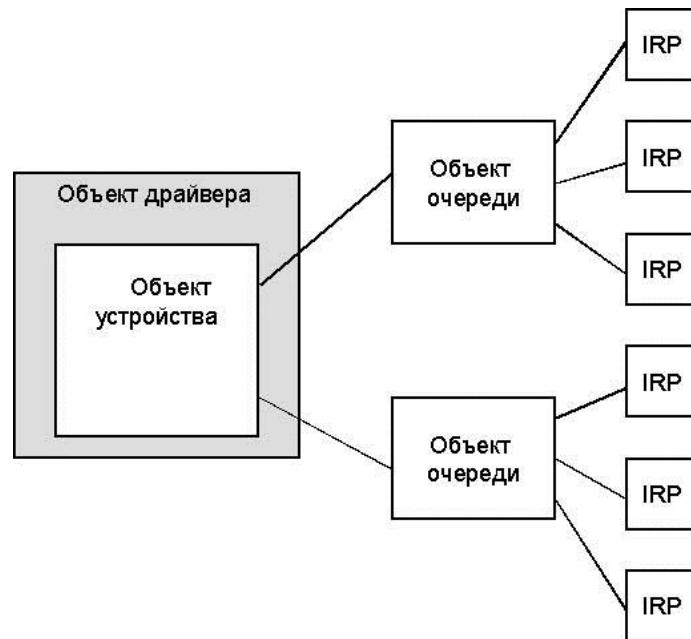


Рисунок 2.3 – Объект очереди, внедренный в объект драйвера.

Объекты, внедренные в объект устройства, представлены в классе KDeviceQueue. Его методы не только реализуют манипуляцию с очередью, но и решают более интеллектуальные задачи. Например, есть метод, смысл которого может быть описан таким образом: "Если устройство сейчас обрабатывает запрос и занято, то помести новый запрос в очередь, иначе немедленно начни его обработку". Подобные методы сильно облегчают задачу буферизации запросов для объекта устройства. Но возможна и другая ситуация: устройство может одновременно обрабатывать запросы разного вида. К примеру, наше устройство – это дуплексный канал связи. Оно одновременно может и принимать, и передавать информацию. Если мы будем использовать для буферизации всех одну очередь, то такой подход является неэффективным. Поэтому система позволяет объектам устройств создавать дополнительные объекты очередей. Они реализованы в классе KDriverManagedQueue.

Эти классы имеют методы, сходные с методами класса KDeviceQueue. Впрочем, ситуации, когда следует применять более одной очереди для буферизации запросов, встречаются не так уж и часто.

Обработка запросов на прерывание при помощи DriverWorks

В контексте данного руководства будем считать, что прерывание (Interrupt) – асинхронный аппаратный сигнал, который возникает, когда периферийному устройству необходимы ресурсы процессора. Термин "асинхронный" означает, что прерывание возникает в произвольные моменты времени (если вообще возникает). Прерывание заставляет процессор прервать выполнение программы, сохранить свое состояние, обработать поступивший запрос (вызывается процедура обработки прерывания, Interrupt Service Routine, ISR) и возобновить выполнение прерванной программы. При этом останавливаются все остальные процессы и потоки операционной системы вне зависимости от их приоритета.

Для того чтобы удовлетворять разнообразным требованиям, возникающим при работе разнообразных устройств и программ на различных типах компьютеров, операционная система предлагает концепцию уровня запроса на прерывание (Interrupt Request Level), IRQ. Всего существует 32 IRQ для данного процессора, пронумерованных от 0 до 31. При этом 0 – самый низкий приоритет, 31 – самый высокий.

Таблица 2.1 – Уровни IRQ.

31	Сбой работы шины
29	Сбой в цепи питания
28	Запрос от другого процессора (в многопроцессорной системе)
	Прерывания, доступные устройствам В/В
2	Выполнение DPC
1	Исключение защиты (page fault)
0	Passive level

Для катастрофических событий операционная система резервирует самые приоритетные прерывания (31 – 29). Для программных прерываний – прерывания с самым низким приоритетом (2 – 1). `PassiveLevel` – обычный режим работы драйвера. `IRQL`, предоставляемые для работы системных устройств, находятся где-то посередине нумерации уровней. О том, как эти прерывания сопрягаются с архитектурой компьютера, заботится HAL.

Естественно, в любой момент процессор может обрабатывать только один запрос прерывания. Обработка поступившего прерывания прервется только в том случае, если поступит прерывание с более высоким приоритетом.

При проектировании процедуры обработки прерывания следует минимизировать время, которое будет затрачено на его обработку. В противном случае процессор будет слишком долго обрабатывать прерывание, и ни один процесс не сможет возобновить свою работу. Когда вызывается ISR, первое, что она должна сделать, это сообщить оборудованию, что запрос на прерывание получен и обработан. После этого можно завершать обработку прерывания. Но как тогда обработать данные, поступившие от устройства, если мы сразу же завершим обработку прерывания? Для этого введен механизм вызова отложенных процедур (Deferred Procedure Call, DPC). Перед завершением работы ISR следует вызвать отложенную процедуру (DPC). DPC начнет выполняться, как только процессор освободится от обработки прерываний. `DriverWorks` предоставляет класс `KDeferredCall`, в котором инкапсулируются данные и методы, необходимые для использования механизма DPC.

`DriverWorks` инкапсулирует все функции, необходимые для обработки прерываний, в классе `KInterrupt`. Экземпляр класса `KInterrupt` должен быть создан, как свойство в классе устройства. Пусть в нашем

случае класс устройства называется MyDevice, объект класса KInterrupt – m_TheInterrupt. Далее в классе устройства описывается функция ISR:

```
BOOLEAN MyDevice::TheIsr(void);
```

Далее, в методе OnStartDevice следует добавить код для привязки ISR к устройству:

```
status =  
m_TheInterrupt.InitializeAndConnect(pAssignedResource,  
    Isr, Context, 0, FALSE);
```

где Context – значение без типа (void), передаваемое ISR.

 Isr – адрес ISR, процедуры обработки прерываний.

Теперь осталось только добавить в конструктор следующий код:

```
VOID MyDevice::MyDevice(void)  
{  
    . . .  
    status =  
m_TheInterrupt.InitializeAndConnect(pAssignedResource,  
    LinkTo(Isr), this, 0, FALSE );  
    . . .  
}
```

Для отключения ISR следует вызвать метод Disconnect().

Естественно, данное описание не претендует быть полным описанием такой важной темы, как обработка прерываний и связанные с ней проблемы. Но в примере драйвера, описываемом ниже, отсутствует реакция на прерывания, а не упомянуть о них нельзя. Для более подробного обзора темы прерываний и DPC следует обратиться к документации DriverWorks или DDK.

Объекты для управления оборудованием

Объект устройства управляет его работой при помощи специальных объектов, управляющих работой оборудования – портами ввода-вывода, прерываниями, памятью, контроллерами ПДП. Драйвер создает эти объекты для представления физических параметров устройства.

Большинство периферийных устройств находятся на шинах компьютера. В современном компьютере есть несколько шин. Обычно процессор, внешняя кэш-память, и оперативная память находятся на высокоскоростной шине, архитектура которой специфична для данного типа процессора. Шина процессора соединена мостом со стандартной скоростной шиной, на которой находятся контроллеры дисплея, некоторые скоростные устройства. Архитектура этой шины может быть процессорно-независимой. Пример такой шины – PCI. Эта шина также может быть соединена мостом со вторичной локальной шиной, часто более медленной. На ней могут находиться контроллеры дисковых накопителей, сетевых адаптеров и т.п.

Периферийные устройства обычно имеют "на борту" регистры и диапазоны адресов памяти, при помощи которых реализуется интерфейс устройства с системой. Но добраться до них не так просто: процессор ведь физически использует другие механизмы для обращения к своим "родным" портам ввода-вывода и оперативной памяти. Для того чтобы обратиться к памяти и портам устройства, находящегося на локальной шине, процессор должен выполнить отображение (mapping) адресного пространства процессора и той шины, где находится наше устройство. В результате этой операции к участку памяти, физически находящейся в устройстве, можно обращаться, как к участку оперативной памяти процессора. При таком обращении процессор переадресует запрос локальной шине. Но тут следует вспомнить об особенностях архитектуры Windows (да и практически любой современной операционной системы): система поддерживает механизм виртуальной памяти! Пользовательские приложения теперь работают в своем адресном пространстве, а система, в том числе и драйверы, – в своем. Куда же будет отображена память устройства?

Ответ прост. Можно отобразить диапазон адресов устройства, как на адресное пространство системы, так и на адресное пространство пользовательского процесса. Соответственно различаться будет и способ доступа к памяти устройства из приложения пользователя: в первом случае буфер с данными для записи или чтения будет передаваться драйверу из приложения, а в драйвере эти данные будут пересылаться устройству. Во втором случае приложение будет писать и читать данные в выделенный ему участок памяти, который находится в адресном пространстве процесса. Какой механизм выбрать – дело разработчика драйвера.

Объекты, представляющие адресное пространство периферийных устройств, представлены классами `KPeripheralAdress`, `KIoRange`, `KMemoryRange`, `KIoregister`, `KMemoryRegister`. `KPeripheralAdress` является базовым классом для большинства остальных классов управления диапазонами памяти и портов ввода-вывода. Сам класс `KPeripheralAdress` в основном, не используется. Используются, в основном, следующие его подклассы:

- `KIoRange` – диапазон адресов ввода-вывода. Данный класс отображает диапазон адресов портов В/В из адресного пространства какой-либо из шин в адресное пространство процессора. При использовании класса `KIoRange` можно читать и записывать в порты 8, 16, и 32-битные значения.
- `KIoRegister` является альтернативным путем доступа к портам ввода-вывода. В виде экземпляра `KIoRegister` может быть представлен отдельный порт-ввода вывода в диапазоне адресов. Фактически, `KIoRange` – это несколько экземпляров класса `KIoRegister`, объединенных в массив. Создать экземпляр `KIoRegister` можно, используя как стандартный конструктор, так и используя оператор `[]` класса `KIoRange`, например:

```
KIoRange m_range;
```

...

```
KIoRegister m_reg = m_range[6];
```

...

Применение KIoRegister упростит процесс программирования и улучшает читабельность программы.

- KMemoryRange используется для отображения диапазона адресов памяти из адресного пространства шины в адресное пространство процессора (адресное пространство системы). После того, как память будет отображена, драйвер должен использовать методы доступа к памяти, позволяющие оперировать 8, 16 и 32- битными значениями.
- KMemoryRegister аналогичен KIoRegister, за исключением того, что в данном случае он представляет из себя отдельную ячейку памяти в адресном пространстве устройства.
- KMemoryToProcessMap используется для отображения диапазона адресов памяти шины в адресное пространство пользовательского процесса. Это может оказаться очень удобным: пользователь может напрямую общаться с памятью устройства в программе, как с обычным буфером. Впрочем, такое отображение следует применять с большой осторожностью: возможна ситуация, когда пользователь запустит несколько экземпляров программы, и все они начнут работать с памятью устройства одновременно. Вряд ли стоит объяснять, к чему это может привести.

Стоит отметить, что немалая часть устройств может общаться со своей памятью только словами. Длина слова зависит от устройства, и может колебаться в широких пределах. Обычно для PCI-устройств – 32 бита.

В документации настоятельно рекомендуется использовать только указанные классы для управления оборудованием. Это связано с

возможной переносимостью драйвера на другие платформы. При использовании этих классов, которые, в свою очередь, используют функции DDK для доступа к оборудованию, процесс портирования пройдет безболезненно, так как для доступа к устройству будет использован HAL. Если же программист будет пытаться управлять устройствами самостоятельно, то драйвер придется переписывать при переносе на другую платформу.

Есть еще одна причина, по которой стоит использовать эти классы: ведь с ними разрабатывать драйвер намного проще!

Объекты синхронизации

Как и все Windows-программы, драйверы являются частью многозадачной операционной системы, в которой выполняется множество процессов и потоков. Драйвер, как и программа, также может содержать несколько потоков. При этом, естественно, возникает проблема синхронизации работы этих потоков, совместного доступа к данным и т.п. Особенно актуальной эта проблема становится в многопроцессорной системе. Windows 2000 предназначена для работы в многопроцессорных системах, и если пренебречь синхронизацией при разработке драйвера, то это может повлечь за собой неприятные последствия.

Для решения задач синхронизации WDM (и, соответственно, DriverWorks) предлагает различные средства. Простейшим из объектов синхронизации является защелка (Spin Lock), представленная классом KSpinLock. Принцип действия защелки очень прост: чтобы запретить любому другому потоку в системе доступ к данным, нужно вызывать метод Lock защелки. Любой поток, пытающийся получить доступ к заблокированным данным, уснет. Чтобы снять блокировку, нужно вызвать метод Unlock.

Класс диспетчера KDispatcherObject является суперклассом для нескольких важных классов синхронизации. Эти классы управляют

планировщиком Windows и позволяют синхронизировать как работу драйверов, так и работу приложения пользователя и драйвера. Все классы, порожденные от KDispatcherObject, имеют два важных отличия:

- с объектом диспетчера связана логическая переменная – флажок, который может находиться в двух состояниях: сигнализировать (TRUE) и молчать (FALSE),
- если поток вызовет метод Wait диспетчера, он приостановится до тех пор, пока диспетчер не перейдет в состояние "сигнализирует".

При работе с объектами диспетчера и его подклассами следует иметь в виду, что нельзя блокировать поток при обработке прерывания. Последствия будут фатальными.

Подклассы класса KDispatcherObject

KEvent – используется для синхронизации работы потоков. Kevent почти не отличается от объекта диспетчера.

KSemaphore инкапсулирует системный объект семафора. Семафор отличается от объекта события тем, что имеет счетчик. Семафор сигнализирует в том случае, если счетчик больше нуля. Семафоры могут быть полезны, например, при управлении несколькими потоками.

KTimer – таймер. При создании таймера его флажок находится в состоянии "молчит". Временной интервал таймера задается функцией Set с точностью до 100 нс. На практике таймер устойчиво работает с временем ожидания ≥ 10 мс. Когда пройдет указанный промежуток времени, таймер перейдет в состояние "сигнализирует". Подклассом Ktimer является класс KTimedCallBack. В нем по истечении промежутка времени выполняется вызов отложенной процедуры (DPC).

KSystemThread позволяет создать новый поток в драйвере. Потоки в драйвере используются в разных целях. В основном это – поллинг медленных устройств и работа на многопроцессорных системах. Для запуска потока следует создать функцию, которая станет функцией потока

и вызвать метод Start. Для уничтожения потока – метод Terminate. При работе с потоками можно использовать все упомянутые выше классы синхронизации.

Дополнительные классы

DriverWorks предоставляет дополнительные классы для нужд программиста. Это классы очередей, списков, стеков; классы файлов и Unicode-строк; классы синхронизации.

Списки представлены классами KList, KInterlockedList, KInterruptSafeList. Они представляют шаблоны двунаправленных списков и стандартные методы для вставки, удаления и добавления элементов. Различаются эти классы методами синхронизации. KList не содержит никаких методов синхронизации и защиты данных. KInterlockedList использует защелки (spin locks) для защиты внутренних связей в списке. KInterruptSafeList использует присоединенный объект прерывания для защиты связей. По аналогичному принципу работают шаблоны классов FIFO (стек): Kfifo, KLockableFifo, KInterruptSafeFifo. Класс KFile инкапсулирует методы для работы с файлами. Этот класс позволяет читать и записывать данные в файл а также изменять атрибуты файлов. Для представления Unicode – строк используется класс KUstring. Методы данного класса позволяют выполнять сравнение, конкатенацию, доступ к символам строки и разнообразные преобразования типа.

Связь драйвера с приложением пользователя

Очень важен еще один вопрос, связанный с драйверами: как именно с нашим объектом устройства может связаться приложение или другой драйвер? Большинство устройств в системе именованы, хотя теоретически допускается существование неименованных (anonymous) устройств. Связь с устройством можно установить двумя методами:

- при помощи GUID,
- при помощи символической ссылки.

GUID (Globally Unique Identifier, глобально уникальный идентификатор) – 16-байтное уникальное число. GUID используются для идентификации в системе драйверов, COM-объектов и т.п. В идеале, во всем мире не может быть двух одинаковых GUID, поэтому GUID может быть абсолютно уникальным идентификатором драйвера. GUID генерируется на основе текущей даты, времени и номера сетевого адаптера, если такой присутствует, и обычно указывается в заголовочном файле класса устройства и программы, которая хочет связаться с ним приблизительно таким образом:

```
#define MyDevice_CLASS_GUID \  
    { 0xff779f4c, 0x8b57, 0x4a65, { 0x85, 0xc4, 0xc8, 0xad, 0x7a, \  
        0x56, 0x64, 0xa6 } }
```

Символическая ссылка (symbolic link) похожа на путь к файлу и в тексте программы имеет вид:

```
char *sLinkName = "\\.\MyDevice";
```


Если отбросить лишние символы бэкслэша, необходимые для соблюдения синтаксиса C++, то символическая ссылка оказывается строкой `\\.\MyDevice`. Чтобы понять принцип работы символической ссылки, следует знать, что в операционной системе есть системный каталог различных объектов, которые присутствуют в системе: драйверов, устройств, объектов событий, семафоров и т.п. Символическая ссылка – специфический тип объекта, который обеспечивает доступ к другим системным объектам. Специальный подкаталог системного каталога зарезервирован для символических ссылок на другие объекты операционной системы. Программа пользователя может обратиться к этим символическим ссылкам при помощи функций API.

Как же следует проектировать интерфейс с драйвером? Следует использовать GUID или символическую ссылку?

Идентификация драйвера при помощи GUID считается более правильной. Специальные алгоритмы гарантируют, что GUID будет действительно уникальным. А кто мешает разработчику, находящемуся на другом конце света, также создать устройство с той же ссылкой на него `\\.\MyDevice?` Вообще-то, никто. Но с другой стороны, с написанной на понятном английском языке ссылкой гораздо проще обращаться, особенно на этапе разработки драйвера, чем с длинным и непонятным GUID. Так что, вероятно, на этапе разработки и отладки драйвера для интерфейса драйвера с приложением лучше использовать символическую ссылку, а для коммерческой версии драйвера – GUID.

2.2 Использование Driver Wizard

Процесс разработки драйвера при помощи DriverStudio во многом напоминает разработку приложения в среде Visual C++ [6]. Создание проекта происходит при помощи мастера DriverWizard, похожего на мастер Visual C++. Мастер вызывается или из главного меню (Пуск ⇒ Программы ⇒ DriverStudio ⇒ DriverWorks ⇒ DriverWizard) или из среды Visual C++ при помощи пункта меню DriverStudio ⇒ DriverWizard.

Программе DriverWizard соответствует иконка .

Далее при работе мастера появляется серия диалоговых окон, куда пользователь должен ввести данные, необходимые для формирования скелета драйвера.

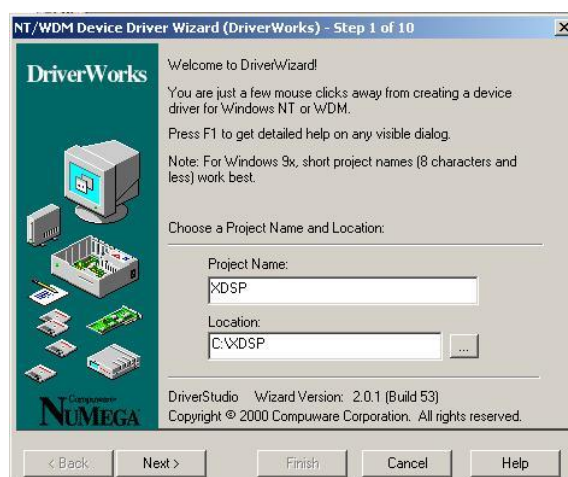


Рисунок 2.4 – Первый шаг DriverWizard

На первом шаге создания драйвера необходимо ввести имя проекта (в нашем случае – XDSP [6]) и директорию для проекта. После этого – нажать на кнопку Next, чтобы перейти к следующему шагу.



Рисунок 2.5 – Второй шаг DriverWizard

На втором шаге следует выбрать архитектуру, по которой будет разрабатываться драйвер: Windows NT 4.0 (которая сейчас практически не используется) или WDM, которую нам и следует выбрать.

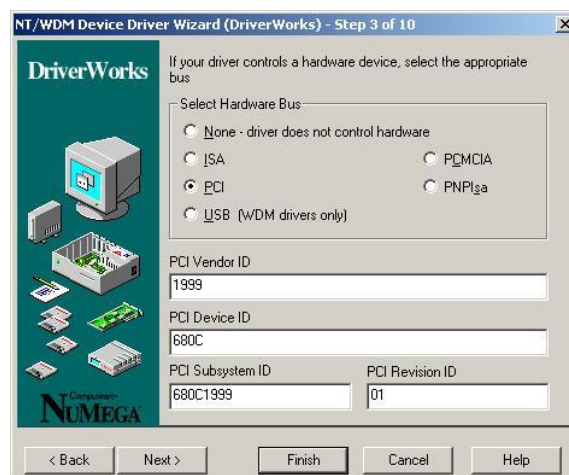


Рисунок 2.6 – Третий шаг DriverWizard

На третьем шаге выберем шину, на которой располагается устройство, которое будет контролировать драйвер. Если это устройство будет подключаться к порту компьютера, например к параллельному, надо выбрать None, driver does not control any hardware. Если же устройство будет располагаться на одной из шин компьютера, например на PCI, надо задать дополнительные параметры. В случае PCI устройства надо указать следующие параметры:

- код производителя (PCI Vendor ID) – четырехзначное шестнадцатеричное число, которое однозначно идентифицирует производителя устройства. Пусть в нашем случае оно будет равно 1999,
- код устройства (PCI Device ID) – также четырехзначное шестнадцатеричное число, которое однозначно идентифицирует устройство нашего производителя. Пусть в нашем случае это будет 680C,
- номер подсистемы PCI. Обычно имеет вид код устройства + код производителя. В нашем случае – 680C1999,
- номер версии устройства (PCI Revision ID) – номер версии устройства. В нашем случае 01.

Эти коды весьма важны: по ним система будет находить драйвер для устройства. Эти же коды аппаратно прошиты в PCI-карточке. И если коды, заданные в драйвере (если быть точным, то они задаются не в самом файле драйвера, а в инсталляционном скрипте – inf-файле), не совпадут с кодами в PCI-устройстве, то драйвер не установится.



Рисунок 2.7 – Четвертый шаг DriverWizard

На четвертом шаге мастера необходимо задать имена, которые DriverWizard присвоит файлу C++, который содержит класс драйвера, и самому классу драйвера (Driver Class).



Рисунок 2.8 – Пятый шаг DriverWizard

На пятом шаге следует указать, какие функции должен выполнять драйвер. Это могут быть:

- чтение (read) – обработка запросов на чтение,
- запись (write) – обработка запросов на запись,
- сброс (flush) – обычно это сброс буфера обмена с устройством,
- управление устройством (device control) – обработка других запросов,
- внутреннее управление устройством (internal device control) – обработка запросов от других драйверов устройств.

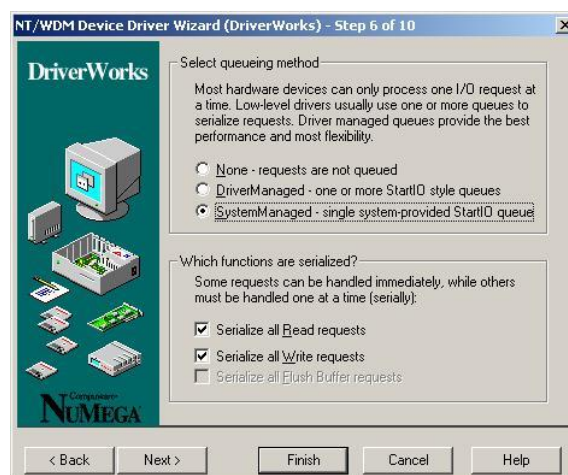


Рисунок 2.9 – Шестой шаг DriverWizard

На шестом шаге DriverWizard задает вопросы о способе обработки запросов. Опция Select queueing method выбирает, каким образом будут буферизироваться запросы на ввод-вывод:

- None – запросы не буферизируются в очереди. Эту опцию лучше не выбирать,
- DriverManaged – драйвер содержит одну или более одной очередей, в которой сохраняются запросы на ввод-вывод, пришедшие от других драйверов или системы,
- SystemManaged – драйвер использует только одну очередь сообщений.

Надо также выбрать, будут ли буферизироваться запросы на чтение и запись. Устройство может одновременно выполнять какую-то одну операцию, например, только чтение или только запись, или может выполнять несколько операций сразу. Чтобы гарантировать нормальную работу устройства в этом случае, следует буферизировать (Serialize) поступающие запросы на чтение и запись, помещая их в очередь. Установка флажков Seralize all Read requests и Serialize all Write requests позволяет буферизировать все запросы на чтение и запись, поступающие в объект устройства.

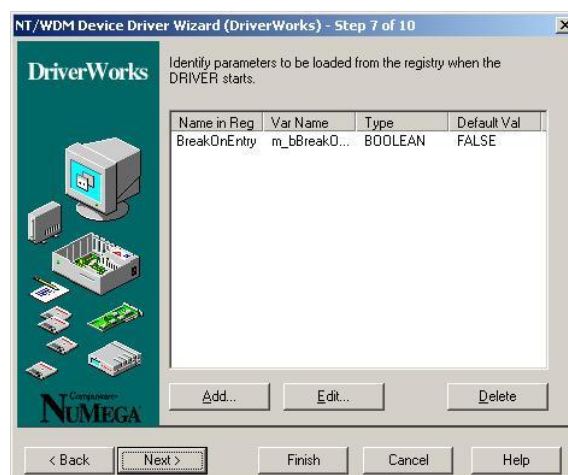


Рисунок 2.10 – Седьмой шаг DriverWizard.

На седьмом шаге предлагается задать параметры, которые драйвер будет загружать из реестра Windows при старте, когда система загружается. При этом задается параметр реестра, имя переменной, куда сохраняется его значение, тип данного параметра и его значение по умолчанию. Если не менять настройки, то во время загрузки драйвер читает из реестра параметр BreakOnEntry типа boolean, сохраняет его значение в переменной m_BreakOnEntry. Значение по умолчанию для параметра – false. Обычно m_BreakOnEntry используется в отладочных целях.

Запись и считывание параметров из реестра позволяет драйверу задавать различные конфигурационные параметры, сохранять данные, необходимые для его запуска или работы.

При помощи кнопок Add, Edit и Delete можно соответственно добавлять, редактировать и удалять параметры.

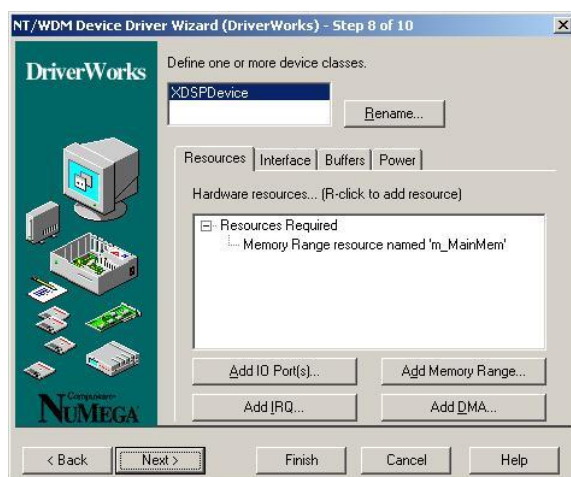


Рисунок 2.11 – Восьмой шаг DriverWizard.

Восьмой шаг DriverWizard – один из самых важных моментов в разработке драйвера PCI – устройства при помощи DriverWorks. Поэтому окно мастера несет огромное количество информации и элементов управления.

На данном шаге предлагается изменить имена классов устройства для данного драйвера. В списке в верхней части окна следует выбирать класс устройства, который следует переименовать, и, нажав на кнопку Rename, можно задать новое имя класса устройства.

Окно DriverWizard также содержит несколько вкладок:

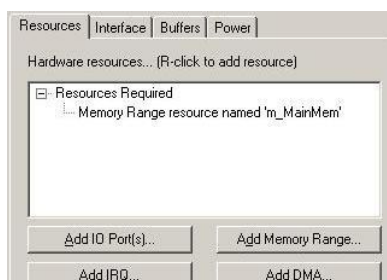


Рисунок 2.12 – вкладка Resource

Вкладка Resource. На ней определяются основные аппаратные ресурсы, которые есть в устройстве и которые будет контролировать этот драйвер. В их числе адреса памяти, диапазоны портов ввода-вывода, линии запроса на прерывание и линии прямого доступа к памяти (DMA), которые необходимы для работы драйвера. Задать ресурсы можно при помощи кнопок в нижней части вкладки.

Например, задать диапазон памяти, которую несет "на борту" устройство, можно, нажав на кнопку Add Memory Range. При этом выводится диалоговое окно, куда следует ввести сведения о новом диапазоне адресов памяти: имя объекта класса KMemoryRange, который будет контролировать этот диапазон адресов, адрес базового регистра в PCI – заголовке (PCI header) данного устройства, который определяет этот диапазон адресов, а также параметры доступа для данной памяти: только чтение (Read Only), только запись (Write Only) и полный доступ (Read/Write). Также можно еще задать опции разделения доступа (Share options). Эти опции позволяют разделять доступ к ресурсу: к нему можно обращаться только из класса данного устройства (Exclusive to this device), из любой части драйвера (Shareable within this driver) или из любого драйвера в системе (Shareable system wide). Впрочем, для разработки простых драйверов эти опции являются бесполезными и изменять их не стоит. В нашем случае мы создаем диапазон адресов памяти с именем m_MainMemoryRange, определяемый нулевым базовым регистром в PCI – header'e, с полным доступом.

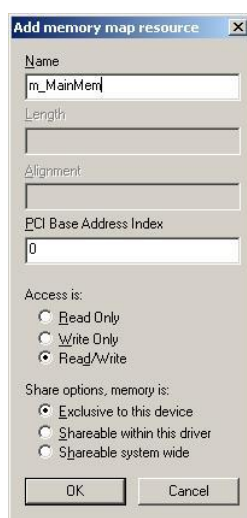


Рисунок 2.13 – Задание диапазона адресов памяти

Аналогично можно задать параметры портов ввода-вывода и линий ПДП. Параметры запросов прерываний сложнее: тут можно дать указание

DriverWizard'у создать шаблоны для классов ISR, DPC и их функций (Make ISR/DPC class functions).

Если в процессе задания ресурсов возникла ошибка, или необходимо внести какие-либо изменения, следует щелкнуть по названию ресурса в окне правой клавишей мыши. Появится контекстное меню, в котором надо выбрать пункт Delete, чтобы удалить ресурс или Edit – редактировать его.

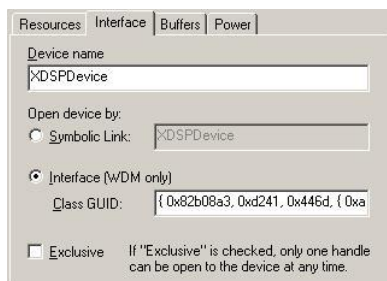


Рисунок 2.14 – Вкладка Interface.

На вкладке Interface задается способ, каким образом будет осуществляться связь программы или библиотеки DLL с драйвером.

Надежным способом является связь при помощи GUID класса. GUID – уникальный номер, который однозначно идентифицирует какой-либо объект системы. При помощи GUID идентифицируются не только драйверы, но и COM – интерфейсы и пр.

Другим способом реализации интерфейса является символическая ссылка. Это более естественный путь, так как просто указать имя класса устройства – гораздо проще, чем указывать непонятного вида GUID.

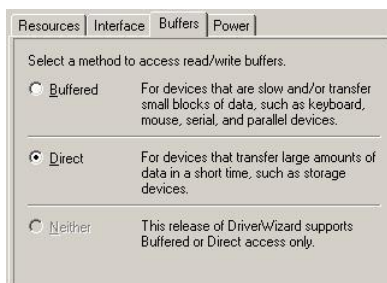


Рисунок 2.15 – Вкладка Buffers.

На вкладке Buffers определяется метод, каким образом буферизируются запросы к устройству.

Буферизированный (buffered) метод – пригоден для устройств типа мыши, клавиатуры, которые передают небольшие объемы данных за короткий промежуток времени. Прямой (direct) метод – используется при пересылке больших объемов информации за короткий промежуток времени, например, при обращении к дисководу.



Рисунок 2.116 – Вкладка Power.

При создании WDM-драйвера необходимо задать способ управления энергопотреблением. При помощи флажка Управлять энергопотреблением этого устройства (Manage power for this device) можно создать в драйвере методы управления энергопотреблением нашего устройства. В нашем простом случае мы не будем этого делать.

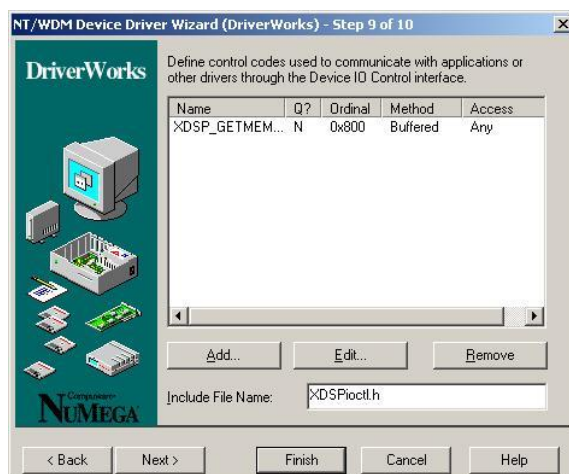


Рисунок 2.17 – Девятый шаг DriverWizard.

Конечно, для более-менее сложного драйвера устройства будет недостаточно двух запросов на чтение и запись. На девятом шаге можно задать коды управления драйвером устройства. Код управления (Device IO control code, IOCTL) просто представляет собой число, которое передается драйверу. Коды управления в драйвере обрабатываются специальной

функцией. В ответ на каждый код драйвер выполняет какое-либо действие. Например, в нашем случае объект устройства будет возвращать количество памяти, которое имеет PCI-карточка. Для этого зададим код управления `XDSP_GetMemSize`. Для этого нажмем на кнопку `Add`, появится диалоговое окно `Edit IO Control Code` (редактирование кода управления).

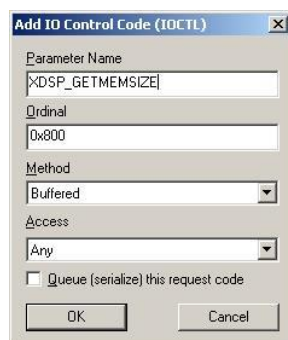


Рисунок 2.18 – Задание кода управления драйвером.

При задании кода управления устройством нужно указать имя кода в понятном программисту виде, метод общения с устройством (прямой или буферизированный). Также задается порядковый номер кода (`Ordinal`) – число, являющееся его уникальным номером. Числа, меньшие `0x800` используются для стандартных кодов, таких, как чтение, запись и т.п.

Запросы `IOCTL` также можно буферизировать, подобно запросам на чтение и запись. Для этого надо установить флажок `Queue (serialize) this request code`.

Внизу окна мастера указано имя заголовочного файла, в котором будут храниться коды управления устройством. В нашем случае это `XDSPioctl.h`. Ненужные коды управления устройством можно удалить, нажав на кнопку `Remove` или редактировать, нажав кнопку `Edit`.



Рисунок 2.19 – Десятый шаг DriverWizard.

Одним из достоинств DriverWorks является то, что DriverWizard сразу создает консольное приложение для тестирования работоспособности драйвера. Конечно, такое тестирование бывает неполным и примитивным, но позволяет оценить, правильно ли работает драйвер и работает ли он вообще. Для того, чтобы DriverWizard создал такое приложение, нужно установить флажок Create test console application (создать консольное приложение для тестирования) и указать его имя. Также можно задать опции отладки. Они необходимы при отладке драйвера средствами DriverStudio. При написании простых драйверов эти опции, скорее всего, не понадобятся.

Пройдя все эти шаги, следует нажать на кнопку Finish. Появится окошко, которое содержит сведения о каталоге с файлами проекта создаваемого драйвера, для чего предназначен каждый файл. После нажатия на кнопку OK DriverWizard сгенерирует все файлы нашего драйвера, приложения для тестирования и предложит открыть проект в Visual C++.

2.3 Компиляция и установка драйвера

Проект, сгенерированный DriverWizard, находится в каталоге XDSP. В этом каталоге расположены файлы рабочего пространства (Workspace) VC++: XDSP.dsw, XDSP.ncd и XDSP.opt и два каталога – sys и exe. Здесь же находится файл XDSPioctl.h. В нем описаны управляющие коды, используемые при обращении к драйверу с помощью функции DeviceIOControl.

В каталоге sys находится сгенерированный DriverWizard скелет драйвера и все необходимые для компиляции файлы. В нашем случае, имеем файлы:

Function.h

заголовочный файл, предназначенный для определения функций, входящих в драйвер;

Makefile, Sources

файлы с информацией, необходимой для компиляции проекта в VC++.

XDSP.h , XDSP.cpp

файлы, содержащие класс драйвера.

XDSP.plg, XDSP.dsp

проект VC++;

XDSP.inf

скрипт для инсталляции драйвера;

XDSP.rc

файл ресурсов проекта. В основном, содержит информацию о разработчике драйвера.

XDSPDevice.cpp, XDSPDevice.h

файлы, содержащие класс устройства.

В каталоге exe находится исходный код консольного приложения TextXDSP, предназначенного для тестирования работы драйвера. При помощи него можно убедиться, правильно ли установлен драйвер в

системе, а иногда даже проверить, как он работает. Хотя для более-менее сложного драйвера придется писать программу тестирования отдельно. В каталоге присутствуют файлы:

Makefile.Sources

файлы с информацией, необходимой для компиляции проекта в VC++.

Test_XDSP.plg, Test_XDSP.dsp

проект VC++;

Test_XDSP.cpp

исходный текст приложения.

Теперь можно открыть проект драйвера в среде VC++ и посмотреть, что же мы имеем. Для этого надо запустить VC++ и открыть проект, используя команду File \Rightarrow Open Workspace. В появившемся диалоговом окне открытия файла выберите файл XDSP.dsw. Если все вышеописанные действия выполнены правильно, то проект откроется в среде VC++. Для того, чтобы проект скомпилировался правильно, следует установить переменные среды DriverStudio. Для этого нужно выбрать пункт меню DriverStudio \Rightarrow Driver Build Settings: На экране появится диалоговое окно установки переменных среды:

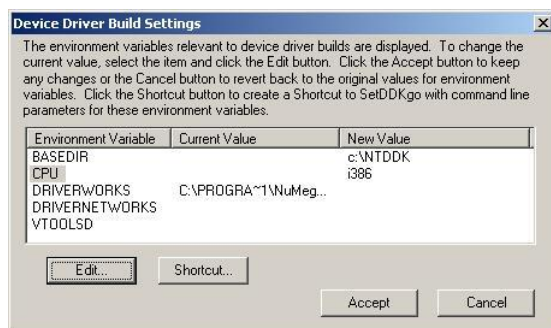


Рисунок 2.20 – Установка значений переменных среды.

Для компиляции драйвера важны две переменные:

 CPU – определяет архитектуру процессора, под которую компилируется драйвер. Не стоит забывать, что Win2000 может работать

на платформах i386 (классические процессоры Intel), IA64 (64-разрядные процессоры Intel) и Alpha. В нашем случае надо установить значение i386.

 BASEDIR – путь к пакету DDK, установленному в системе. Для того, чтобы изменить значение одной из этих переменных, надо нажать кнопку Edit: диалогового окна. Появится окно установки значений переменной.

Установив требуемое значение, нажмите кнопку Set. Чтобы закрыть окно – Exit. Задав переменные среды, нажмите кнопку Ассерт. Теперь можно компилировать проекты.

Драйвер может быть скомпилирован в двух конфигурациях: Checked и Free.

Checked – отладочный вариант драйвера. Такой драйвер несет в себе информацию для отладки. Естественно, что для отладки драйверов непригодны обыкновенные отладчики, входящие в комплект сред VC++, Delphi и т.п. Все они работают в 3-м кольце привилегий процессора и даже не догадываются, какие драйвера есть в системе. Для отладки драйверов применяются специальные отладчики, работающие в режиме ядра операционной системы. В качестве отладчика лучше всего использовать SoftIce, поставляемый с DriverStudio.

Free-драйвер не несет отладочную информацию. Активную конфигурацию можно выставить при помощи пункта меню Build ⇒ Set Active Configuration:

Особенность сгенерированного DriverWizard рабочего пространства состоит в том, что оно содержит два проекта: XDSP и Test_XDSP. Как нетрудно догадаться, XDSP – это проект драйвера, а Test_XDSP – приложения тестирования. Информация о проектах выводится в окне Workspace среды VC++.

В каждый отдельный момент времени можно компилировать только активный проект. Имя активного проекта выводится жирным шрифтом.

Сделать активным другой проект просто: надо щелкнуть на его названии правой клавишей мыши и в выпавшем контекстном меню выбрать пункт Set as Active Project (Сделать активным проектом).

Теперь можно выполнять компиляцию проекта. Если в процессе компиляции появляются сообщения об ошибках – значит, вы не совсем точно следовали инструкциям, изложенным выше: или не скомпилировали библиотеки DriverWorks, или не установили переменные среды.

После компиляции драйвера следует скомпилировать тестовое приложение Test_XDSP. Оно должно скомпилироваться без каких-либо проблем.

Если все операции прошли гладко – можно себя поздравить: драйвер готов к работе. Хотя, естественно, он не выполняет никаких разумных действий. Теперь можно протестировать наш драйвер.

После компиляции мы получили файл драйвера XDSP.sys. Он находится в каталоге ../XDSP/sys/obj/i386. В этом каталоге будут находиться скомпилированные DriverStudio драйверы. Но для инсталляции кроме самого драйвера еще нужен скрипт XDSP.inf. Он обычно находится в самом каталоге XDSP.

Итак, для установки драйвера в системе предполагается наличие в системе PCI – карточки XDSP-680. После установки карточки (или перепрограммирования ее из среды Foundation) следует перезагрузить компьютер. При загрузке компьютер обнаружит новое устройство и потребует предоставить драйвер для него. Если же не потребует – значит в системе есть более ранняя версия драйвера. Для этого надо открыть список устройств, установленных на компьютере и обновить драйвер для устройства. Для этого надо указать путь к скрипту xdsp.inf и к файлу драйвера xdsp.sys.

Если же Вы разрабатываете драйвер, который не управляет каким-либо устройством или это устройство не является PnP – необходимо

просто установить драйвер стандартными средствами Windows: Пуск \Rightarrow Настройка \Rightarrow Панель управления \Rightarrow Установка оборудования. Когда Windows выведет полный список типов устройств и спросит, какое устройство Вы хотите установить, выберите свой тип устройства.

После того, как драйвер будет установлен, нужно будет проверить его функционирование. Запустите скомпилированный файл `test_xdsp.exe` с параметрами `test xdsp r 32` (команда прочитать 32 байта из устройства).

Должно появиться сообщение, похожее на это:

```
C:\XDSP\exe\objchk\i386>Test XDSP.exe r 32
```

Test application Test XDSP starting...

```
Device found, handle open.
```

Reading from device - 0 bytes read from device (32 requested).

$$\begin{array}{ccccccccc} - & / & - & / & - & / & - & / & - \\ - & / & - & / & - & / & - & / & \end{array}$$

— / — / — / — / — / — /

В данном случае приложение установило связь с драйвером и прочитало из него 32 байта. Функция чтения в драйвере не определена, поэтому, естественно, драйвер вернет абракадабру. Если же будет получено сообщение вида

```
C:\...Projects\XDSPdrv\exe\objchk\i386>Test XDSP.exe r 32
```

Test application Test XDSP starting...

```
ERROR opening device: (2) returned from CreateFile
```

Exiting...

– то приложение не смогло установить связь с драйвером. Следует попробовать переустановить драйвер.

2.4 Наращивание функциональных возможностей драйвера

Рассмотрим подробно текст драйвера, сгенерированного DriverWizard и внесем в него необходимые изменения.

В проекте присутствуют всего два класса:

`XDSP`

класс драйвера;

`XDSPDevice`

класс устройства.

Также есть несколько глобальных функций и переменных:

`PNPMinorFunctionName` – возвращает строку с текстовым названием кода функции IOCTL. Эта функция используется при отладке, когда надо перевести числовое значение кода IOCTL в строку с его названием.

`POOLTAG DefaultPoolTag('PSDX')` – используется совместно с `BoundsChecker` для отслеживания возможных ситуаций переполнения буфера и утечек памяти.

`KTrace t("XDSPdrv")` – глобальный объект трассировки драйвера. Этот объект используется для вывода сообщений трассировки при работе драйвера. Использование объекта трассировки аналогично использованию класса `iostream` в C++. Вывод отладочных сообщений производится при помощи оператора `<<`. Примеры использования объекта трассировки неоднократно встречаются в тексте драйвера, например:

```
t << "m_bBreakOnEntry loaded from registry,  
resulting value: [" << m_bBreakOnEntry << "]\n";
```

В данном примере объект трассировки используется для вывода строки `"m_bBreakOnEntry loaded from registry, resulting value: ["` и значения логической переменной `m_bBreakOnEntry`. Все сообщения трассировки можно прочитать в отладчике `SoftIce`.

Начнем анализ текста драйвера с класса `XDSP` (класс драйвера). В строке 31 при помощи макроса `DECLARE_DRIVER_CLASS`

декларируется класс драйвера XDSP. Далее следует метод DriverEntry, который вызывается при инициализации драйвера:

```
NTSTATUS XDSPdrv::DriverEntry(PUNICODE_STRING RegistryPath)
//В строке RegistryPath содержится ключ реестра, в котором
// система хранит информацию о драйвере.
{
    //Далее выводится трассировочное сообщение, информирующее
    // о вызове метода DriverEntry:
    t << "In DriverEntry\n";
    //После этого драйвер создает объект Params класса
    // KRegistryKey и считывает данные из
    //реестра для этого драйвера:
    KRegistryKey Params(RegistryPath, L"Parameters");
    //Далее производится проверка на успех:
    if ( NT_SUCCESS(Params.LastError()) )
    {
        //Текст, заключенный в марос препроцессора DBG будет
        откомпилирован
        // только в отладочной версии
        //драйвера.
        #if DBG
            ULONG bBreakOnEntry = FALSE;
            // Читается значение переменной BreakOnEntry реестра:
            Params.QueryValue(L"BreakOnEntry", &bBreakOnEntry);
            // Если она принимает значение true, то инициировать
            // точку останова в отладчике.
            if (bBreakOnEntry) DbgBreakPoint();
        #endif
        //Загрузить остальные параметры реестра.
        LoadRegistryParameters(Params);
    }
    m_Unit = 0;
    //Вернуть успех
    return STATUS_SUCCESS;
}
```

```
}
```

Метод LoadRegistryParameters загружает из реестра все остальные параметры, необходимые для драйвера. Впрочем, в нашем драйвере таковых нет, и поэтому функция не выполняет никаких полезных действий (просто загружает значение переменной m_bBreakOnEntry).

```
void XDSPdrv::LoadRegistryParameters(KRegistryKey &Params)
{
    m_bBreakOnEntry = FALSE;
    Params.QueryValue(L"BreakOnEntry", &m_bBreakOnEntry);
    t << "m_bBreakOnEntry loaded from registry, resulting
value:
    [" << m_bBreakOnEntry << "]\n";
}
```

На этом заканчивается секция инициализации драйвера. Далее следует метод AddDevice. Он вызывается, когда система обнаруживает устройство, за которое отвечает драйвер (обычно это происходит при загрузке драйвера). В метод система передает указатель на физический объект устройства (Physical Device Object, PDO). Этот объект представляет собой некий блок информации о физическом устройстве, который используется операционной системой. Данный метод создает объект устройства XDSPDevice. С точки зрения системы, создается функциональный объект устройства (Functional Device Object, FDO).

```
NTSTATUS XDSPdrv::AddDevice(PDEVICE_OBJECT Pdo)
{
    t << "AddDevice called\n";
    //Здесь вызывается конструктор класса XDSPDevice.
    XDSPdrvDevice * pDevice = new (
        static_cast(KUnitizedName(L"XDSPdrvDevice",
m_Unit)),
        FILE_DEVICE_UNKNOWN,
        static_cast(KUnitizedName(L"XDSPdrvDevice",
m_Unit)),
```

```

        0,
        DO_DIRECT_IO
    )

    XDSPDevice(Pdo, m_Unit);
//m_Unit - количество таких устройств в системе.
if (pDevice == NULL)
    //Не удалось создать объект устройства. Похоже, произошла
какая-то ошибка.
{
    t << "Error creating device XDSPdrvDevice"
        << (ULONG) m_Unit << EOL;
    return STATUS_INSUFFICIENT_RESOURCES;
}
//Получить статус создания устройства.
NTSTATUS status = pDevice->ConstructorStatus();
if ( !NT_SUCCESS(status) )
    //Похоже, устройство создано, но неудачно; произошла
ошибка.
{
    t << "Error constructing device XDSPdrvDevice"
        << (ULONG) m_Unit << " status " << (ULONG) status <<
EOL;
    delete pDevice;
}
else
{
    m_Unit++;    //Устройство создано удачно
}
//Вернуть статус устройства.
return status;
}

```

Все. Работа объекта драйвера на этом окончена. Как мы можем видеть, объект драйвера практически не выполняет каких-либо функций управления аппаратурой, но он жизненно необходим для правильной

инициализации драйвера. В нашем случае НЕ ТРЕБУЕТСЯ вносить какие-либо изменения в текст, сформированный DriverWizard.

Основным классом драйвера является класс устройства. Класс устройства XDSPdrvDevice является подклассом класса KpnpDevice. Конструктор получает два параметра: указатель на PDO и номер драйвера в системе.

```
XDSPdrvDevice::XDSPdrvDevice(PDEVICE_OBJECT Pdo, ULONG Unit) :
    KPnpDevice(Pdo, NULL)
{
    t    <<    "Entering    XDSPdrvDevice::XDSPdrvDevice
(constructor)\n";
//Здесь проверяется код ошибки, которую вернул конструктор
суперкласса. В случае
//успешного создания объекта базового класса значение
переменной m_ConstructorStatus
//будет NT_SUCCESS.
if ( ! NT_SUCCESS(m_ConstructorStatus) )
{
    //Ошибка в создании объекта устройства
    return;
}
//Запомнить номер драйвера
m_Unit = Unit;
//Инициализация устройства нижнего уровня. В роли устройства
нижнего уровня в нашем
//драйвере выступает PDO. Но в случае стека драйверов в
качестве устройства нижнего
//уровня может выступать объект устройства другого драйвера.
    m_Lower.Initialize(this, Pdo);
// Установить объект нижнего уровня для нашего драйвера.
    SetLowerDevice(&m_Lower);
    // Установить стандартную политику PnP для данного
устройства.
```

```

        SetPnpPolicy();
    }

```

Порядок вызова методов `m_Lower.Initialize(this, Pdo)`, `SetLowerDevice(&m_Lower)` и `SetPnpPolicy()` является жизненно важным. Его нарушение может вызвать серьезные сбои в работе драйвера. Не стоит редактировать текст конструктора, сгенерированный DriverWizard.

Деструктор объекта устройства не выполняет никаких действий. Но для сложных драйверов, когда создаются системные потоки, разнообразные объекты синхронизации и выделяется память, то все созданные объекты должны быть уничтожены в деструкторе. В нашем простейшем случае не стоит вносить изменения в текст деструктора.

```

XDSPdrvDevice::~~XDSPdrvDevice()
{
    t << "Entering XDSPdrvDevice::~~XDSPdrvDevice()
(destructor)\n";
}

```

Метод `DefaultPnp` – виртуальная функция, которая должна быть переопределена любым объектом устройства. Эта обработчик по умолчанию для IRP-пакета, у которого старший код функции (`major function code`) равен `IRP_MJ_PNP`. Драйвер обрабатывает некоторые из таких пакетов, у которых младший код функции равен `IRP_MN_STOP_DEVICE`, `IRP_MN_START_DEVICE` и т.п. (см. ниже) также при помощи виртуальных функций. Но те пакеты, которые не обрабатываются объектом устройства, передаются этой функции. Она ничего с ними не делает, а просто передает их устройству нижнего уровня (если такое есть, конечно). Не стоит изменять текст этой функции.

```

NTSTATUS XDSPdrvDevice::DefaultPnp(KIrp I)
{
    t << "Entering XDSPdrvDevice::DefaultPnp with IRP minor
function="
    << PNPMajorFunctionName(I.MinorFunction()) << EOL;
}

```

```

        I.ForceReuseOfCurrentStackLocationInCalldown();
        return m_Lower.PnpCall(this, I);
    }

```

Метод `SystemControl` выполняет похожую функцию для IRP-пакетов, у которых старший код функции `IRP_MJ_SYSTEM_CONTROL`. Он также является виртуальной функцией и не выполняет никаких полезных действий, а просто передает IRP-пакет устройству нижнего уровня. Что-то менять в тексте этого метода надо только в том случае, если наше устройство является WMI-провайдером.

```

NTSTATUS XDSPdrvDevice::SystemControl(KIrp I)
{
    t << "Entering XDSPdrvDevice::SystemControl\n";

    I.ForceReuseOfCurrentStackLocationInCalldown();
    return m_Lower.PnpCall(this, I);
}

```

Метод `Invalidate` вызывается, когда устройство тем или иным образом завершает свою работу: из функций `OnStopDevice`, `OnRemoveDevice` а также при всевозможных ошибках. Метод `Invalidate` объекта устройства также вызывается из деструктора. Его можно вызывать несколько раз – не произойдет ничего страшного; но в методах `Invalidate` нет никакой защиты от реентерабельности. Т.е. если при работе метода `Invalidate` возникает какая-либо ошибка и из-за этого `Invalidate` должен будет вызваться снова, то ни `DriverWorks`, ни ОС Windows не станут этому мешать. Разработчик должен сам предусмотреть такую возможность и принять меры, чтобы подобная ситуация не привела к нехорошим последствиям.

В методе `Invalidate` объекта устройства вызываются методы `Invalidate` всех ресурсов, которые использует драйвер: областей памяти, регистров, контроллеров DMA и т.п. При этом выполняется процедура, обратная

процедуре инициализации: освобождаются все ресурсы, используемые объектом, закрываются все его хэндлы, но сам объект не уничтожается и может быть проинициализирован снова. В нашем простом случае нет смысла что-либо корректировать в тексте этого метода – DriverWizard все сделал за нас. Еще бы, ведь наше устройство имеет только один ресурс – диапазон адресов памяти. Но при проектировании более сложных драйверов следует обращать внимание на данный метод. Если разработчик добавляет какие-либо системные ресурсы вручную, то он должен включить соответствующий код в метод Invalidate.

```
VOID XDSPdrvDevice::Invalidate()  
{  
    //Вызвать метод Invalidate для диапазона адресов памяти.  
    m_MainMem.Invalidate();  
}
```

Далее следует виртуальная функция OnStartDevice. Она вызывается при приходе IRP- пакета со старшим кодом IRP_MJ_PNP и кодом подфункции IRP_MN_START_DEVICE. Обычно это происходит при старте драйвера после выполнения всех необходимых проверок и инициализаций. В этой функции драйвер инициализирует физическое устройство и приводит его в рабочее состояние. Также здесь драйвер получает список ресурсов, которые имеются в устройстве. На основе этого списка ресурсов выполняется их инициализация. Хотя мы не вносим изменений в данную функцию, но нельзя не отметить ее огромную важность. Именно в данной функции выполняется инициализация устройства и получение списка его ресурсов. По-другому мы их получить никак не можем, т.к. имеем дело с PnP устройством, для которого система распределяет ресурсы самостоятельно.

```
NTSTATUS XDSPdrvDevice::OnStartDevice(KIrp I)  
{  
    t << "Entering XDSPdrvDevice::OnStartDevice\n";  
    NTSTATUS status = STATUS_SUCCESS;  
    I.Information() = 0;
```



```

//Здесь драйвер получает список ресурсов устройства
    PCM_RESOURCE_LIST pResListRaw = I.AllocatedResources();
    PCM_RESOURCE_LIST pResListTranslated =
I.TranslatedResources();

// Наше устройство является PCI - карточкой и в своем
конфигурационном поле содержит
//базовые адреса диапазонов памяти и портов ввода-вывода.
Получаем эти данные
    KPciConfiguration PciConfig(m_Lower.TopOfStack());

// Инициализируем каждый диапазон отображения адресов. Теперь,
после инициализации,
//базовый адрес отображенного диапазона в виртуальном адресном
пространстве
//процессора может быть получен при помощи вызова метода
Base(). Физический адрес на
//шине адреса ЦП - при помощи CpuPhysicalAddress().

    status = m_MainMem.Initialize(
        pResListTranslated,
        pResListRaw,
        PciConfig.BaseAddressIndexToOrdinal(0)
    );

    if (!NT_SUCCESS(status))
    {
        //Неудача при инициализации области памяти
        Invalidate();
        return status;
    }

//Сюда можно добавить код, выполняющий необходимую
инициализацию, специфичную для

```

```
//этого устройства
    return status;
}
```

Виртуальная функция OnStopDevice вызывается при остановке устройства. В этом случае система посылает драйверу IRP с старшим кодом IRP_MJ_PNP и кодом подфункции IRP_MN_STOP_DEVICE. Драйвер должен освободить все используемые ресурсы.

```
NTSTATUS XDSPdrvDevice::OnStopDevice(KIrp I)
```

```
{
    NTSTATUS status = STATUS_SUCCESS;
    t << "Entering XDSPdrvDevice::OnStopDevice\n";
    //Освободить ресурсы устройства
    Invalidate();
```

```
// Здесь добавляется код, специфичный для данного устройства.
```

```
    return status;
}
```

Виртуальная функция OnRemoveDevice вызывается при извлечении устройства из системы. При этом системная политика PnP сама позаботится об удалении PDO.

```
NTSTATUS XDSPdrvDevice::OnRemoveDevice(KIrp I)
```

```
{
    t << "Entering XDSPdrvDevice::OnRemoveDevice\n";
    // Освободить ресурсы устройства
    Invalidate();
```

```
// Здесь добавляется код, специфичный для данного устройства.
```

```
    return STATUS_SUCCESS;
}
```

Иногда бывает необходимо отменить обработку IRP, уже поставленного в очередь (такие запросы иногда посылает диспетчер ввода-вывода). Когда такая ситуация может возникнуть?

Представим такую ситуацию: в приложении пользователя поток послал нашему драйверу запрос на ввод-вывод и завершил свою работу. А IRP-пакет попал в очередь запросов и терпеливо ждет своей очереди на обработку. Конечно же, обработка такого "бесхозного" IRP-пакета должна быть отменена. Для этого драйвером поддерживается целый механизм отмены обработки запросов. В Win2000 DDK подробно описано, почему ЛЮБОЙ драйвер должен поддерживать этот механизм. Это связано, в основном, с проблемами надежности и устойчивости работы системы. Ведь сбой в драйвере – это, практически, сбой в ядре операционной системы.

В классе KPNPDevice механизм отмены запроса реализован при помощи метода CancelQueuedIrp.

```
VOID XDSPdrvDevice::CancelQueuedIrp(KIrp I)
{
    //Получаем очередь IRP-пакетов этого устройства.
    KDeviceQueue dq(DeviceQueue());

    //Проверить, является ли IRP, который должен быть отменен, тем
    пакетом, который должен
    //быть обработан.
    if ( (PIRP)I == CurrentIrp() )
    {
        //Уничтожить пакет.
        CurrentIrp() = NULL;
    }
    //При вызове метода CancelQueuedIrp устанавливается глобальная
    системная
    //защелка (SpinLock). Теперь следует ее сбросить.
    CancelSpinLock::Release(I.CancelIrql());
}
```

```

        t << "IRP canceled " << I << EOL;
        I.Information() = 0;
        I.Status() = STATUS_CANCELLED;
//Обработать следующий пакет.
        PnpNextIrp(I);
    }
//Удалить из очереди пакет. Если это удалось, то функция
вернет true.
    else if (dq.RemoveSpecificEntry(I))
    {
// Это удалось. Теперь сбрасываем защелку.
        CancelSpinLock::Release(I.CancelIrql());
        t << "IRP canceled " << I << EOL;
        I.Information() = 0;
        I.PnpComplete(this, STATUS_CANCELLED);
    }
    else
    {
//Неудача. Сбрасываем защелку.
        CancelSpinLock::Release(I.CancelIrql());
    }
}

```

Метод `StartIo` является виртуальной функцией. Она вызывается системой, когда драйвер готов обрабатывать следующий запрос в очереди. Это чрезвычайно важная функция: она является диспетчером всех запросов на ввод-вывод, поступающих к нашему драйверу. В функции вызываются обработчики запросов на чтение, запись а также обработчики вызовов `IOCTL`. К счастью, умный `DriverWizard` генерирует скелет функции автоматически и вносить изменения в нее в нашем простом случае не требуется. В принципе, в эту функцию можно ввести какие-то дополнительные проверки `IRP`-пакетов.

```

VOID XDSPdrvDevice::StartIo(KIrp I)

```

```

{
    t << "Entering StartIo, " << I << EOL;

    //Здесь надо проверить, отменен этот запрос или нет. Это производится
    // при помощи вызова метода TestAndSetCancelRoutine. Также этот метод
    // устанавливает новую функцию отмены пакетов, если это необходимо.
    // Адрес новой функции передается вторым параметром. Если он равен
    // NULL, то вызывается старая функция. Если пакет должен быть отменен,
    // функция вернет FALSE.
    if ( !I.TestAndSetCancelRoutine(
        LinkTo(CancelQueuedIrp),
        NULL,
        CurrentIrp()) )
    {

        //Пакет отменен.
        return;
    }

    // Начать обработку запроса.
    // Выбрать необходимую функцию
    switch (I.MajorFunction())
    {
        case IRP_MJ_READ:
            //Чтение
            SerialRead(I);
            break;
        case IRP_MJ_WRITE:
            //Запись
            SerialWrite(I);
            break;
        case IRP_MJ_DEVICE_CONTROL:
            //IOCTL
            switch (I.IoctlCode())

```

```

        {
            default:
                //Мы обрабатываем пакет, который не должен быть
                // обработан. Поэтому просто выходим.
                ASSERT(FALSE);
                break;
        }
        break;
    default:
        // Драйвер занес в очередь какой-то непонятный пакет,
        //он не должен быть обработан.
        ASSERT(FALSE);
        PnpNextIrp(I);
        break;
    }
}

```

Метод Create вызывается, когда пользовательское приложение пытается установить связь с драйвером при помощи вызова API CreateFile(). Обычно этот запрос обрабатывается в нашем объекте устройства и нет смысла пересылать запрос устройству нижнего уровня.

```

NTSTATUS XDSPdrvDevice::Create(KIrp I)
{
    NTSTATUS status;

    t << "Entering XDSPdrvDevice::Create, " << I << EOL;

    //Здесь можно вставить код пользователя, который должен быть
    //вызван при установлении
    //приложением связи с устройством.

    status = I.PnpComplete(this, STATUS_SUCCESS,
    IO_NO_INCREMENT);
}

```

```

        t << "XDSPdrvDevice::Create Status " << (ULONG)status <<
EOL;
        return status;
}

```

Аналогично метод Close вызывается при разрыве связи приложения с драйвером.

```

NTSTATUS XDSPdrvDevice::Close(KIrp I)
{
    NTSTATUS status;
    t << "Entering XDSPdrvDevice::Close, " << I << EOL;

    //Здесь можно вставить код пользователя, который должен быть
    //вызван при разрыве
    //приложением связи с устройством.

    status = I.PnpComplete(this, STATUS_SUCCESS,
IO_NO_INCREMENT);
    t << "XDSPdrvDevice::Close Status " << (ULONG)status <<
EOL;
    return status;
}

```

В этих методах можно ввести проверки каких-либо условий.

Написание драйвера завершено. Может сложиться впечатление, что DriverWizard может практически все и написание драйвера – очень простая задача. Но не следует забывать, что рассмотренный драйвер –простейшая демонстрационная программа, которая практически не выполняет никаких полезных действий. Если бы драйвер был написан с использованием пакета DDK, то он бы имел практически ту же структуру и почти тот же код (правда, не объектно-ориентированный). Но в таком случае весь драйвер пришлось бы писать вручную, а DriverWizard генерирует скелет

драйвера автоматически. Это сильно облегчает процесс разработки драйвера, позволяя программисту не заботиться о написании скелета драйвера и предохраняя его от возможных ошибок.

2.5 Контрольные вопросы к разделу

1. Что такое DDK?
2. Из каких этапов состоит создание драйвера?
3. Для чего используется программный пакет DriverStudio?
4. Что необходимо для работы в пакете DriverStudio?
5. Из каких модулей состоит пакет DriverStudio?
6. Для чего нужен модуль DriverWorks?
7. Для чего нужен модуль VtoolsD?
8. Для чего нужен модуль SoftIce?
9. Каков состав системы классов DriverWorks?
10. Имеется ли в системе классов DriverWorks иерархия?
11. Что такое объект драйвера?
12. Для чего нужен класс KRegistryKey?
13. Что такое объект запроса на ввод-вывод IRP?
14. Кто создает IRP?
15. Что такое объект устройства?
16. Для чего нужны классы KDevice и KPnPDevice?
17. Для чего нужен класс KMemoryRange?
18. Что такое стек устройств?
19. Что такое объект очереди?
20. Для чего нужна буферизация запросов?
21. Как обрабатываются запросы на прерывания в DriverWorks?
22. Что такое концепция уровней прерывания?
23. Что делает процедура обработки прерывания?
24. Для чего нужен вызов отложенных процедур?
25. Для чего нужен класс KIoRange?
26. Для чего нужен класс KIoRegister?
27. Для чего нужен класс KMemoryToProcessMap?
28. Для чего нужен класс KDispatcherObject?

Литература

1. В. Пирогов. Ассемблер для Windows. — 3-е изд. — СПб.: БХВ-Петербург, 2005.
2. М. Руссинович, Д. Соломон. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows X P, Windows 2000. — 4-е изд. — М.: Русская Редакция, Питер, 2005.
3. Валерий Яковлев. Написание пользовательской DLL доступа к универсальному OPC-серверу Fastwel // Современные технологии автоматизации. 2005. № 3. С. 74-81.
4. Валерий Яковлев. Основы написания драйвера уровня ядра для ОС Windows 2000, XP и XP Embedded // Современные технологии автоматизации. 2006. № 2. С. 68-75.
5. Рошин А.В. Организация ввода-вывода. Часть 2. Драйверы для WINDOWS NT. Учебное пособие/ А.В.Рошин. — М.: МГУПИ, 2006. — 112 с.: ил.
6. Тарво А. Использование NuMega DriverStudio для написания WDM-драйверов. Интернет-ресурс: <http://worldcpp.vinograd.ru/vspp/syst5.php>

7.

Учебное пособие

Рощин Алексей Васильевич

Организация ввода-вывода.

Часть 3. Основы написания драйверов уровня ядра для операционной
системы Windows NT5
Учебное пособие.

Подписано к печати __.__.2009 г.

Формат 60×84 1/16.

Объем 6,0 п.л. Тираж 100 экз. Заказ № __

Отпечатано в типографии Московской государственной академии
приборостроения и информатики
107076, Москва, ул. Стромынка 20.