Министерство образования и науки Российской Федерации

ФБГОУ ВПО «Кубанский государственный технологический университет»

Кафедра вычислительной техники и АСУ

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Методические указания по выполнению курсовой работы
для студентов всех форм обучения специальности
230101 Вычислительные машины, комплексы, системы и сети

Краснодар
2012

Составители: канд. техн. наук, доцент А.Г. Мурлин;
ст. преподаватель А.Г. Волик

УДК 681.31(031)

Системное программное обеспечение: метод. указания по выполнению курсового проекта для студентов всех форм обучения специальности 230105 Программное обеспечение вычислительной техники и автоматизированных систем / Сост.: А. Г. Мурлин, А. Г. Волик; Кубан. гос. технол. ун-т. Каф. вычислительной техники и АСУ. – Краснодар.: Изд. КубГТУ, 2012 – 84 с.

Содержат требования к оформлению и содержанию курсового проекта, примерный перечень тем курсовых проектов и основные сведения по разработке драйверов для ОС Windows и Linux. Приведены примеры простейших Legacy-драйверов, WDM-драйверов, модулей ядра Linux и рекомендуемая литература.

Библиогр.: 20 назв. Прил. 2.

Печатается по решению методического совета Кубанского государственного технологического университета

Рецензенты:
зав. кафедрой ВТиАСУ КубГТУ, д-р техн. наук, проф. В. И. Ключко;
руководитель отдела телекоммуникаций ООО Информационный центр «Консультант», канд. техн. наук Н. Ф. Григорьев

Содержание

Введение

Курсовая работа выполняется студентами специальности 230101 в седьмом семестре.

Целью курсовой работы является закрепление основ и углубление знаний в области разработки системного программного обеспечения, получения дополнительных практических навыков в создании программного продукта системного назначения.

При выполнении курсовой работы студент самостоятельно выполняет все этапы создания программного продукта, от постановки задачи до практической реализации, сопровождающейся документацией и инструкциями по его использованию.

В процессе выполнения работы студент получает навыки самостоятельного использования специальной литературы, каталогов, справочников, стандартов. Тематика заданий на курсовую работу, приведенных в данных методических указаниях, может быть дополнена, расширена, увязана с решением актуальных научно-исследовательских задач, выполняемых на кафедре.

1 Нормативные ссылки

ГОСТ Р 1.5-2004 Стандарты национальные Российской Федерации. Правила построения, изложения, оформления и обозначения
ГОСТ 2.104-68 ЕСКД. Основные надписи
ГОСТ 7.1-2003 Библиографическое описание документа. Общие требования и правила составления
ГОСТ 7.80-2000 СИБИД. Библиографическая запись. Заголовок. Общие требования и правила составления
ГОСТ 7.82-2001 СИБИД. Библиографическая запись. Библиографическое описание электронных ресурсов. Общие требования и правила составления
ГОСТ 19.001-77 ЕСПД. Общие положения
ГОСТ 19.104-78 ЕСПД. Основные надписи
ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам
ГОСТ 19.401-78 ЕСПД. Текст программы. Требования к содержанию и оформлению
ГОСТ 19.402-78 ЕСПД. Описание программы
ГОСТ 19.404-79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению

2 Требования к курсовому проекту

Тема курсового проекта выдается каждому студенту индивидуально. В коллективных работах, в которых принимают участие два и более студентов, четко определяется объем и характер работы каждого студента.

В задании формулируется задача, метод ее решения.

Курсовой проект состоит из пояснительной записки, к которой прилагается электронный носитель с исходными текстами и программами.

Пояснительная записка должна содержать:

- титульный лист (приложение А);
- задание на курсовое проектирование (приложение Б);
- реферат по ГОСТ 7.9 (количество страниц ПЗ, количество таблиц, рисунков, схем, программ приложений, ключевые слова, краткая характеристика и результаты работы);
- содержание;
- нормативные ссылки;
- введение;
- основную часть, включающую:
  а) спецификации задачи;
  б) формулировку задачи;
  в) описание методов и подходов, используемых при решении;
  г) описание примененных в работе методов программирования;

д) тексты программ (листинги) по ГОСТ 19.401;
е) описание программы по ГОСТ 19.402;
ж) результаты машинного тестирования программы;

- заключение (основные результаты работы);
- список использованных источников (по ГОСТ 7.80, ГОСТ 7.82).

Пояснительная записка должна быть оформлена по ГОСТ Р 1.5 и ГОСТ 19.105, ГОСТ 19.404 на листах формата А4 (210 × 297 мм) по ГОСТ 2.301. Все листы следует сброшюровать и пронумеровать. Объем пояснительной записки – не менее 15 - 20 и не более 40 - 50 страниц.

3 Перечень тем курсовых работ

1. Разработка и исследование драйвера AT-клавиатуры для ОС Linux.
2. Разработка и исследование драйвера COM-мыши для ОС Linux.
3. Разработка и исследование драйвера PS/2-мышки для ОС Linux.
4. Разработка и исследование драйвера сенсорной панели для ОС Linux.
5. Разработка и исследование драйвера игрового порта для ОС Linux.
6. Разработка и исследование драйвера FireWire (IEEE1394) для ОС Linux.
7. Разработка и исследование драйвера шины PCI для ОС Linux.
8. Разработка и исследование драйвера шины I2C для ОС Linux.
9. Разработка и исследование драйвера сетевой карты для ОС Linux.
10. Разработка и исследование драйвера IDE-интерфейса для ОС Linux.
11. Разработка и исследование драйвера шины USB для ОС Linux.
12. Разработка и исследование драйвера AT-клавиатуры для ОС Windows.
13. Разработка и исследование драйвера COM-мыши для ОС Windows.
14. Разработка и исследование драйвера PS/2-мышки для ОС Windows.
15. Разработка и исследование драйвера сенсорной панели ОС Windows.
16. Разработка и исследование драйвера игрового порта для ОС Windows.
17. Разработка и исследование драйвера FireWire (IEEE1394) для Windows.
18. Разработка и исследование драйвера шины PCI для ОС Windows.
19. Разработка и исследование драйвера шины I2C для ОС Windows.
20. Разработка и исследование драйвера сетевой карты для ОС Windows.
21. Разработка и исследование драйвера IDE-интерфейса для ОС Windows.
22. Разработка и исследование драйвера шины USB для ОС Windows.
23. Разработка и исследование драйвера виртуального устройства.
24. Разработка однопросмотрового ассемблера.
25. Разработка двухпросмотрового ассемблера.
26. Разработка службы (service) для ОС Windows.
27. Разработка демона (daemon) для ОС Linux.
28. Разработка сетевого приложения для ОС Windows.
29. Разработка сетевого приложения для ОС Linux.
30. Разработка программы редактора связей.

31. Разработка модуля управления памятью для вычислительного комплекса без использования дискового пространства.
32. Разработка модуля управления памятью для вычислительного комплекса с использованием дискового пространства.
33. Разработка программы, расширяющей буфер обмена Windows.
34. Разработка программы загрузчика операционной системы.
35. Разработка виртуальной машины DOS.
36. Разработка программы монитора разделяемого ресурса.
37. Разработка модуля управления организацией работы в критической секции в многозадачной системе.
38. Разработка модуля предотвращения, распознавания и восстановления системы после тупиковых ситуаций.
39. Разработка модуля управления процессами в многозадачной системе.

4 Методика выполнения основных элементов курсового проекта

При выполнении пояснительной записки к курсовому проекту необходимо во введении кратко описать актуальность работы, использованные при разработке программы приемы программирования и инструменты, основные задачи разрабатываемого ПО.

Перед разработкой алгоритмов и программ необходимо ознакомиться с литературой, описывающей методику и технологию разработки соответствующего выбранной тематике программного обеспечения, наметить основные методы, используемые для разработки ПО. Поскольку большинство программ системного назначения работает непосредственно с аппаратным обеспечением компьютера, то необходимо четко представлять архитектуру ПК и работу процессора в различных режимах.

При описании разработанных алгоритмов допускается опускать несущественные детали и подробно описывать наиболее существенные.

Необходимо подробно описать используемые инструментальные программы и библиотеки, а также способы построения исполняемых модулей. Кроме этого следует осветить нюансы взаимодействия с аппаратной частью компьютера, программным обеспечением системного назначения и ядром операционной системы. В пояснительную записку обязательно должен быть включен раздел тестирования разработанного ПО, а также описание способов его установки.

5 Основные теоретические сведения о разработке драйверов

5.1 Общие понятия

В большинстве случаев, под драйвером понимается фрагмент кода операционной системы, который позволяет обращаться к аппаратному обеспечению. В большинстве современных операционных систем принят

способ работы с драйверами как с файлами. Кроме этого существует общий для всех систем механизм воздействия на драйвер, – при помощи IOCTL запросов, – позволяющий выполнять более специфичные операции.

Драйверы устройств, как правило, – наиболее критическая часть программного обеспечения компьютеров.

В большинстве случаев, под драйвером понимается фрагмент кода операционной системы, который позволяет ей обращаться к аппаратному обеспечению. В эту категорию входят как неотъемлемые части компьютера (например, наборы микросхем на материнских платах современных персональных компьютеров), так и вполне автономные устройства (например, внешние модемы).

Концепция драйвера как отдельного сменного модуля оформилась не сразу. Некоторые версии UNIX до сих пор практикуют полную перекомпиляцию ядра при замене какого-либо драйвера, что совершенно не похоже на обращение с драйверами в Linux, Windows и MS DOS. Кстати, именно MS DOS ввела в массовое обращение понятие драйвера, как легко сменяемой насадки, позволяющей моментально (сразу после очередной перезагрузки) повысить качество работы пользователя.

Касаясь характерных черт драйвера (работающего с полномочиями компонента ядра) для разных операционных систем – Windows и Linux – остановимся на трех основных принципах.

В операционных системах MS DOS, Windows, Unix и всех клонах Linux принят способ работы с драйверами как с файлами. То есть при доступе к драйверу используются функции либо совпадающие (лексически), либо весьма похожие на функции для работы с файлами (open, close, read, write, CreateFile...).

Данный порядок неудивителен для Unix-систем, поскольку в них вся действительность воспринимается в виде файлов (что является изначальной концепцией данной ветви операционных систем). Например, директорию (каталог файлов) можно открыть как файл и считывать оттуда блоки данных, соответствующие информации о каждом хранящемся в этой директории файле. В директории /dev/ можно открыть файл, соответствующий мышке и считывать постепенно байты данных, появляющиеся в нем в точном соответствии с ее перемещениями.

В ОС Windows предлагает точно такой же механизм. Для доступа к драйверу из своего приложение пользователь прибегает к помощи функции CreateFile (это могла бы быть функция OpenFile, но это название морально устарело, поскольку использовалась в старых 16-тиразрядных версиях Windows). Правда, имя файла, который предполагается "открыть", выглядит странно, и на жестком диске такого файла отыскать невозможно – он существует лишь в недрах операционной системы и выглядит, например, как "\\\\.\\myDevice". (Операционная система понимает его как сим-

вольную ссылку для идентификации конкретного драйвера, привлекаемого к работе.) И хотя дальнейшие операции, сформулированные создателем пользовательского приложения как вызовы read()-write(), все-таки преобразуются операционной системой в специальные запросы к драйверу, необходимо признать: формально процесс похож на работу с файлом.

Драйверы стали легко заменяемой запасной частью в операционной системе. Если раньше и были различия между продуктами Microsoft и Unix-системами (драйверы в операционных системах Microsoft изначально были "подвижно-сменными", но в UNIX и ранних версиях Linux при их замене надо было заново выполнять перекомпиляцию ядра), то сейчас такие различия исчезли. При сохранении некоторых особенностей инсталляции, драйверы теперь повсеместно могут быть удалены/добавлены в систему редактированием одной записи в специальных системных файлах. Более того, загрузка "по требованию" (по запросу пользовательской программы) становится практически общей чертой Windows/Unix/Linux. Даже операционные системы реального времени, например, QNX также используют методику сменных драйверов.

Концепция существования режима ядра (с большими функциональными возможностями и относительной бесконтрольности) и пользовательского режима (с жестким контролем со стороны системы) присутствует в Windows/Unix/Linux с незапамятных времен. Если внимательно посмотреть на то, как в Linux реализуется драйвер, то увидим, что это всего лишь модуль ядра, который имеет некое (дополнительное) отражение в виде файла в директории /dev/. Если посмотреть теперь на драйвер (режима ядра) в операционной системе Windows, то становится понятно: это не просто драйвер, это возможность войти в режим ядра со своим программным кодом.

При этом существует общий для всех систем механизм воздействия на драйвер – при помощи IOCTL запросов – позволяющий осуществлять более функциональное взаимодействие.

5.2 Разработка драйверов для ОС Windows

5.2.1 Структура драйвера Windows NT (Legacy Driver)

Все драйверы с точки зрения степени привилегированности их кода делятся на две группы: функционирующие в пользовательском режиме и работающие в режиме ядра.

Первые из них, драйверы пользовательского режима, представляют собой обычный программный код, как правило, оформленный в динамически загружаемые библиотеки (DLL). Эти драйверы стеснены в обращении к системным ресурсам и опираются в своей работе на модули режима ядра, с которыми они тесно сотрудничают.

Программирование в режиме ядра имеет свои специфические особенности. В частности, в качестве библиотечных функций (типа привычных malloc и free) в режиме ядра применяются системные вызовы (например, ExAllocatePool и ExFreePool).

Рассмотрим, как организованы драйверы режима ядра и как происходит связь с ними из приложений пользовательского режима на примере простого драйвера Example.sys.

Приведенный ниже код драйвера Example.sys является завершенным драйвером, который готов к компиляции и использованию в операционной системе в качестве тестового примера.

Драйвер – это DLL режима ядра. Он реализован как набор функций, каждая из которых предназначена для реализации отдельного типа обращений к драйверу со стороны Диспетчера ввода/вывода. Экспорт этих функций выполняется путем их регистрации в процедуре, стандартной для всех драйверов, – **DriverEntry**. Драйвер может быть загружен и выгружен, а для выполнения действий по инициализации или освобождению ресурсов драйвер должен зарегистрировать соответствующие рабочие функции.

Все приведенные ниже отрывки кода следует последовательно поместить в один файл (обычно, файл, содержащий описание **DriverEntry**, разработчики называют Init.cpp). Компиляцию драйвера (как чистовую, так и отладочную) рекомендуется выполнять утилитой Build из среды DDK, поскольку иные способы компиляции могут быть источником необъяснимых странностей в поведении драйвера.

Вспомогательная функция **CompleteIrp** реализует действия по завершению обработки IRP пакета с кодом завершения status. Данная функция предназначена для внутренних нужд драйвера и нигде не регистрируется. Параметр info, если он не равен нулю, чаще всего содержит число байт, переданных клиенту (полученных от клиента) драйвера.

Процедура **ReadWrite_IRPhandler** предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_READ/IRP_MJ_WRITE по результатам обращения к драйверу из пользовательских приложений с вызовами read/write или из кода режима ядра с вызовами ZwReadFile или ZwWriteFile. В данном примере наша функция обработки запросов чтения/записи ничего полезного не делает, и ее регистрация выполнена только для демонстрации, как это могло бы быть в более "развитом" драйвере.

Процедура **Create_File_IRPprocessing** предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_CREATE по результатам обращения к драйверу из пользовательских приложений с вызовами CreateFile или из кода режима ядра с вызовами ZwCreateFile. В нашем примере эта функция не выполняет никаких особых действий (хотя можно было бы завести счетчик открытых

дескрипторов и т.п.), однако без регистрации данной процедуры система просто не позволила бы клиенту "открыть" драйвер для работы с ним (хотя сам драйвер мог бы успешно загружаться и стартовать).

Процедура **Close_File_IRPprocessing** предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодом IRP_MJ_CLOSE по результатам обращения к драйверу из пользовательских приложений с вызовами CloseHandle или из кода режима ядра с вызовами ZwClose. В нашем примере эта функция не выполняет никаких особых действий, однако, выполнив регистрацию процедуры открытия файла, мы теперь просто обязаны зарегистрировать процедуру завершения работы клиента с открытым дескриптором. Заметим, что если клиент пользовательского режима забывает закрыть полученный при открытии доступа к драйверу дескриптор, то за него эти запросы выполняет операционная система (впрочем, как и в отношении всех открытых приложениями файлов, когда приложения завершаются без явного закрытия файлов).

Процедура **DeviceControlRoutine** предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодом IRP_MJ_DEVICE_CONTROL по результатам обращения к драйверу из пользовательских приложений с вызовами DeviceIoControl.

В нашем примере это самая важная функция. Она реализует обработку пяти IOCTL запросов:

- IOCTL_PRINT_DEBUG_MESS – выводим отладочное сообщение в окно DebugView.
- IOCTL_CHANGE_IRQL – проводим эксперимент, насколько высоко можно искусственно поднять уровень IRQL в коде драйвера.
- IOCTL_MAKE_SYSTEM_CRASH – проводим эксперимент по "обрушению" операционной системы и пытаемся его предотвратить.
- IOCTL_TOUCH_PORT_378H – проводим эксперимент по обращению к аппаратным ресурсам системы.
- IOCTL_SEND_BYTE_TO_USER – отправляем байт данных в пользовательское приложение.

Эти IOCTL коды являются пользовательскими – они определены с помощью макроса CTL_CODE в файле Driver.h, который является частью данного проекта, и речь о котором пойдет ниже.

Процедура **UnloadRoutine** выполняет завершающую работу перед тем как драйвер будет выгружен.

Листинг 1 – Пример простого драйвера (файл init.cpp)

```
//////////////////////////////////////////////////////////////////
// init.cpp: Инициализация драйвера
// Замечание. Рабочая версия данного драйвера должна быть
// скомпилирована как не-WDM версия. В противном случае - драйвер
```

```
// не сможет корректно загружаться и выгружаться с использованием
// сервисов SCM Менеджера.
/////////////////////////////////////////////////////////////////////
// DriverEntry            Главная точка входа в драйвер
// UnloadRoutine          Процедура выгрузки драйвера
// DeviceControlRoutine   Обработчик DeviceIoControl IRP пакетов
/////////////////////////////////////////////////////////////////////
#include "Driver.h"

// Предварительные объявления функций:
NTSTATUS DeviceControlRoutine( IN PDEVICE_OBJECT fdo, IN PIRP Irp );
VOID     UnloadRoutine(IN PDRIVER_OBJECT DriverObject);
NTSTATUS ReadWrite_IRPhandler( IN PDEVICE_OBJECT fdo, IN PIRP Irp );
NTSTATUS Create_File_IRPprocessing(IN PDEVICE_OBJECT fdo, IN PIRP Irp);
NTSTATUS Close_HandleIRPprocessing(IN PDEVICE_OBJECT fdo, IN PIRP Irp);

// Нехорошо использовать глобальные переменные в драйвере,
// но для простоты примера воспользуемся ...
KSPIN_LOCK MySpinLock;
#pragma code_seg("INIT") // начало секции INIT
/////////////////////////////////////////////////////////////////////
//
// DriverEntry - инициализация драйвера и необходимых объектов
// Аргументы:  указатель на объект драйвера
//             раздел реестра (driver service key) в UNICODE
// Возвращает: STATUS_Xxx
//
extern "C"
NTSTATUS DriverEntry( IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath )
{
   NTSTATUS status = STATUS_SUCCESS;
   PDEVICE_OBJECT  fdo;
   UNICODE_STRING  devName;

   #if DBG
   DbgPrint("=Example= In DriverEntry.");
   DbgPrint("=Example= RegistryPath = %ws.", RegistryPath->Buffer);
   #endif

   // Экспорт точек входа в драйвер (AddDevice объявлять не будем)
   // DriverObject->DriverExtension->AddDevice= OurAddDeviceRoutine;
   DriverObject->DriverUnload = UnloadRoutine;
   DriverObject->MajorFunction[IRP_MJ_CREATE]= Create_File_IRPprocessing;
   DriverObject->MajorFunction[IRP_MJ_CLOSE] = Close_HandleIRPprocessing;
   DriverObject->MajorFunction[IRP_MJ_READ]  = ReadWrite_IRPhandler;
   DriverObject->MajorFunction[IRP_MJ_WRITE] = ReadWrite_IRPhandler;
   DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]= DeviceControlRou-
tine;
   //=================================================
   // Действия по созданию символьной ссылки
   // (эти действия необходимо выполнять в OurAddDeviceRoutine,
```

```
// но, так как драйвер предназначен только для демонстрации,
// эта процедура отсутствует):
RtlInitUnicodeString( &devName, L"\\Device\\EXAMPLE" );

// Создаем Functional Device Object (FDO) и получаем
// указатель на созданный FDO в переменной fdo.
// (В WDM драйвере эту работу также следовало бы выполнять
// в процедуре OurAddDeviceRoutine.) При создании FDO
// будет выделено место и под структуру расширения устройства
// EXAMPLE_DEVICE_EXTENSION (для этого мы передаем в вызов
// ее размер, вычисляемый оператором sizeof):
status = IoCreateDevice(DriverObject,
                        sizeof(EXAMPLE_DEVICE_EXTENSION),
                        &devName, // может быть и NULL
                        FILE_DEVICE_UNKNOWN,
                        0,
                        FALSE, // без эксклюзивного доступа
                        &fdo);
if(!NT_SUCCESS(status)) return status;

// Получаем указатель на область, предназначенную под
// структуру расширение устройства
PEXAMPLE_DEVICE_EXTENSION dx =
    (PEXAMPLE_DEVICE_EXTENSION)fdo->DeviceExtension;
dx->fdo = fdo;  // Сохраняем обратный указатель

// Применяя прием условной компиляции, вводим функцию DbgPrint,
// сообщения которой мы сможем увидеть в окне DebugView, если
// выполним сборку нашего драйвера как checked (отладочную) версию:
#if DBG
DbgPrint("=Example= FDO %X, DevExt=%X.",fdo,dx);
#endif

//=====================================
// Действия по созданию символьной ссылки
// (их нужно было бы делать в OurAddDeviceRoutine, но у нас
// очень простой драйвер):
UNICODE_STRING symLinkName;   // Сформировать символьное имя:
// #define   SYM_LINK_NAME   L"\\??\\Example"
// Такого типа символьные ссылки ^^ проходят только в NT.
// (То есть, если перенести бинарный файл драйвера в
// Windows 98, то пользовательские приложения заведомо
// не смогут открыть файл по такой символьной ссылке.)
// Для того чтобы ссылка работала и в Windows 98, и в NT,
// необходимо поступать следующим образом:
#define SYM_LINK_NAME   L"\\DosDevices\\Example"
RtlInitUnicodeString( &symLinkName, SYM_LINK_NAME );
dx->ustrSymLinkName = symLinkName;

// Создаем символьную ссылку
status = IoCreateSymbolicLink( &symLinkName, &devName );
if (!NT_SUCCESS(status))
```

```
    { // при неудаче v удалить Device Object и вернуть управление
        IoDeleteDevice( fdo );
        return status;
         } // Теперь можно вызывать CreateFile("\\\\.\\Example",...);
           // в пользовательских приложениях

         // Объект спин-блокировки, который будем использовать для
         // разнесения во времени выполнения кода обработчика
         // IOCTL запросов. Инициализируем его:
         KeInitializeSpinLock(&MySpinLock);

         // Снова используем условную компиляцию, чтобы выделить код,
         // компилируемый в отладочной версии и не компилируемый в
         // версии free (релизной):
         #if DBG
         DbgPrint("=Example= DriverEntry successfully completed.");
         #endif
         return status;
}
#pragma code_seg() // end INIT section
//
// CompleteIrp: Устанавливает IoStatus и завершает обработку IRP
// Первый аргумент - указатель на объект нашего FDO.
//
NTSTATUS CompleteIrp( PIRP Irp, NTSTATUS status, ULONG info)
{
   Irp->IoStatus.Status = status;
   Irp->IoStatus.Information = info;
   IoCompleteRequest(Irp,IO_NO_INCREMENT);
   return status;
}
//
// ReadWrite_IRPhandler: Берет на себя обработку запросов
// чтения/записи и завершает обработку IRP вызовом CompleteIrp
// с числом переданных/полученных байт (BytesTxd) равным нулю.
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от Диспетчера ввода/вывода
//
NTSTATUS ReadWrite_IRPhandler( IN PDEVICE_OBJECT fdo, IN PIRP Irp )
{
   ULONG BytesTxd = 0;
   NTSTATUS status = STATUS_SUCCESS; //Завершение с кодом status
   // Задаем печать отладочных сообщений v если сборка отладочная
   #if DBG
   DbgPrint("-Example- in ReadWrite_IRPhandler.");
   #endif
   return CompleteIrp(Irp,status,BytesTxd);
}
//
// Create_File_IRPprocessing: Берет на себя обработку запросов с
// кодом IRP_MJ_CREATE.
```

```
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от Диспетчера В/В
//
NTSTATUS Create_File_IRPprocessing(IN PDEVICE_OBJECT fdo,IN PIRP Irp)
{
   PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);
   // Задаем печать отладочных сообщений - если сборка отладочная
   #if DBG
   DbgPrint("-Example- Create File is %ws",
       &(IrpStack->FileObject->FileName.Buffer));
   #endif
return CompleteIrp(Irp,STATUS_SUCCESS,0); // Успешное завершение
}
//
// Close_File_IRPprocessing: Берет на себя обработку запросов с
// кодом IRP_MJ_CLOSE.
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от Диспетчера ввода/вывода
//
NTSTATUS Close_HandleIRPprocessing(IN PDEVICE_OBJECT fdo,IN PIRP Irp)
{
   #if DBG
   // Задаем печать отладочных сообщений - если сборка отладочная
   DbgPrint("-Example- In Close handler.");
   #endif
return CompleteIrp(Irp,STATUS_SUCCESS,0);// Успешное завершение
}
//
// DeviceControlRoutine: обработчик IRP_MJ_DEVICE_CONTROL запросов
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от Диспетчера ВВ
// Возвращает:  STATUS_XXX
// #define SMALL_VERSION
// В том случае, если не закомментировать верхнюю строчку, будет
// выполнена компиляция версии, в которой будет обрабатываться только
// один тип IOCTL запросов -- IOCTL_MAKE_SYSTEM_CRASH
//
NTSTATUS DeviceControlRoutine( IN PDEVICE_OBJECT fdo, IN PIRP Irp )
{
   NTSTATUS status = STATUS_SUCCESS;
   ULONG BytesTxd =0; // Число переданных/полученных байт (пока 0)
   PIO_STACK_LOCATION IrpStack=IoGetCurrentIrpStackLocation(Irp);

   // Получаем указатель на расширение устройства
   PEXAMPLE_DEVICE_EXTENSION dx =
               (PEXAMPLE_DEVICE_EXTENSION)fdo->DeviceExtension;
   //-------------------------------
   // Выделяем из IRP собственно значение IOCTL кода, по поводу
   // которого случился вызов:
```

```
ULONG ControlCode =
IrpStack->Parameters.DeviceIoControl.IoControlCode;
ULONG method = ControlCode & 0x03;

// Получаем текущее значение уровня IRQL v приоритета,
// на котором выполняется поток (вообще говоря, целое число):
KIRQL irql,
currentIrql = KeGetCurrentIrql();

#if DBG
DbgPrint("-Example- In DeviceControlRoutine (fdo= %X)\n",fdo);
DbgPrint("-Example- DeviceIoControl: IOCTL %x.", ControlCode );
if(currentIrql==PASSIVE_LEVEL)
    DbgPrint("-Example- PASSIVE_LEVEL (val=%d)",currentIrql);
#endif
// Запрашиваем владение объектом спин-блокировки. В данном
// примере не выполняется никаких критичных действий, но,
// вообще говоря, этот прием может быть полезен и даже
// незаменим, если в приведенном ниже коде должны будут
// выполнены манипуляции, которые можно делать только
// эксклюзивно. Пока потоку выделен объект спин-блокировки v
// никакой другой поток не сможет войти в оператор switch:
KeAcquireSpinLock(&MySpinLock,&irql);

// Диспетчеризация по IOCTL кодам:
switch(ControlCode) {

#ifndef SMALL_VERSION
case IOCTL_PRINT_DEBUG_MESS:
{     // Только вводим сообщение и только в отладочной версии
    #if DBG
    DbgPrint("-Example- IOCTL_PRINT_DEBUG_MESS.");
    #endif
    break;
}
case IOCTL_CHANGE_IRQL:
{
    #if DBG
    // Эксперименты по искусственному повышению
    // IRQL v только в отладочной версии!
    DbgPrint("-Example- IOCTL_CHANGE_IRQL.");
    KIRQL dl = DISPATCH_LEVEL, // только для распечатки (2)
    oldIrql,
    newIrql=25; // Новый уровень IRQL (например, 25)
    // Устанавливаем newIrql, сохраняя текущий в oldIrql:
    KeRaiseIrql(newIrql,&oldIrql);
    newIrql=KeGetCurrentIrql(); // Что реально получили?

    DbgPrint("-Example- DISPATCH_LEVEL value =%d",dl);
    DbgPrint("-Example- IRQLs are old=%d new=%d",
                    oldIrql,newIrql);
    KeLowerIrql(oldIrql); // Возвращаем старое значение
```

```
    #endif
    break;
}
#endif // SMALL_VERSION


case IOCTL_MAKE_SYSTEM_CRASH:
{
    int errDetected=0;
    char x = (char)0xFF;

    #if DBG  // Вообще говоря, под NT мы этого уже не увидим:
    DbgPrint("-Example- IOCTL_MAKE_SYSTEM_CRASH.");
    #endif
    // Вызываем системный сбой обращением по нулевому адресу
    __try {
    x = *(char*)0x0L; // ошибочная ситуация
        //^^^^^^^^^^^^ здесь случится сбой NT, но не Win98
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {   // Перехват исключения не работает!
        errDetected=1;
    };
    #if DBG
    DbgPrint("-Example- Value of x is %X.",x);
    if(errDetected)
        DbgPrint("-Example- Except detected in Example driver.");
    #endif
    break;
}

#ifndef SMALL_VERSION
case IOCTL_TOUCH_PORT_378H:
{
    unsigned short ECRegister = 0x378+0x402;
    #if DBG
    DbgPrint("-Example- IOCTL_TOUCH_PORT_378H.");
    #endif
    // Пробуем программно перевести параллельный порт 378,
    // сконфигурированный средствами BIOS как ECP+EPP, в
    // режим EPP.
    _asm  {
        mov dx,ECRegister ;
        xor al,al    ;
        out dx,al    ;    Установить EPP mode 000
        mov al,095h  ;    Биты 7:5 = 100
        out dx,al    ;    Установить EPP mode 100
    }
    // Подобные действия в приложении пользовательского
    // режима под NT обязательно привели бы к аварийной
    // выгрузке приложения с сообщением об ошибке!
    // Практически эти пять строк демонстрируют, что можно
```

```
    // работать с LPT портом под Windows NT!
    break;
}

case IOCTL_SEND_BYTE_TO_USER:
{
    // Размер данных, поступивших от пользователя:
    ULONG InputLength = //только лишь для примера
        IrpStack->Parameters.DeviceIoControl.InputBufferLength;
    // Размер буфера для данных, ожидаемых пользователем
    ULONG OutputLength =
    IrpStack->Parameters.DeviceIoControl.OutputBufferLength;
    #if DBG
    DbgPrint("-Example- Buffer outlength %d",OutputLength);
    #endif

    if(OutputLength < 1)
    {// Если не предоставлен буфер v завершить IRP с ошибкой
        status = STATUS_INVALID_PARAMETER;
        break;
    }
    UCHAR *buff; // unsigned char, привыкаем к новой нотации
    if(method == METHOD_BUFFERED)
    {
        buff = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
        #if DBG
        DbgPrint("-Example- Method : BUFFERED.");
        #endif
    }
    else
        if (method == METHOD_NEITHER)
        {
            buff=(unsigned char*)Irp->UserBuffer;
            #if DBG
            DbgPrint("-Example- Method : NEITHER.");
            #endif
        }
        else
        {
            #if DBG
            DbgPrint("-Example- Method : unsupported.");
            #endif
            status = STATUS_INVALID_DEVICE_REQUEST;
            break;
        }
    #if DBG
    DbgPrint("-Example- Buffer address is %08X",buff);
    #endif
    *buff = 33;      // Запишем чило
    BytesTxd = 1; // Передали 1 байт
    break;
}
```

```
   #endif // SMALL_VERSION
   // Ошибочный запрос (код IOCTL, который не обрабатывается):
   default: status = STATUS_INVALID_DEVICE_REQUEST;
   }
   // Освобождение спин-блокировки
   KeReleaseSpinLock(&MySpinLock, irql);

   #if DBG
   DbgPrint("-Example- DeviceIoControl: %d bytes written.",
(int)BytesTxd);
   #endif

return CompleteIrp(Irp, status, BytesTxd); // Завершение IRP
}
//
// UnloadRoutine: Выгружает драйвер, освобождая оставшиеся объекты
// Вызывается системой, когда необходимо выгрузить драйвер.
// Как и процедура AddDevice, регистрируется иначе чем
// все остальные рабочие процедуры и не получает никаких IRP.
// Arguments:  указатель на объект драйвера
//
#pragma code_seg("PAGE")
// Допускает размещение в странично организованной памяти
//
VOID UnloadRoutine(IN PDRIVER_OBJECT pDriverObject)
{
   PDEVICE_OBJECT  pNextDevObj;
   int i;

   // Задаем печать отладочных сообщений v если сборка отладочная
   #if DBG
   DbgPrint("-Example- In Unload Routine.");
   #endif
   //=========================================================
   // Нижеприведенные  операции в полномасштабном WDM драйвере
   // следовало бы поместить в обработчике IRP_MJ_PNP запросов
   // с субкодом IRP_MN_REMOVE_DEVICE, но в силу простоты
   // драйвера, сделаем это здесь.
   // Проходим по всем объектам устройств, контролируемым
   // драйвером
   pNextDevObj = pDriverObject->DeviceObject;

   for(i=0; pNextDevObj!=NULL; i++)
   {
       PEXAMPLE_DEVICE_EXTENSION dx =
               (PEXAMPLE_DEVICE_EXTENSION)pNextDevObj->DeviceExtension;
       // Удаляем символьную ссылку и уничтожаем FDO:
       UNICODE_STRING *pLinkName = & (dx->ustrSymLinkName);
       // !!! сохраняем указатель:
       pNextDevObj = pNextDevObj->NextDevice;

       #if DBG
```

```
        DbgPrint("-Example- Deleted device (%d) : pointer to FDO = %X.",
                            i,dx->fdo);
        DbgPrint("-Example- Deleted symlink = %ws.", pLinkName->Buffer);
        #endif

        IoDeleteSymbolicLink(pLinkName);
        IoDeleteDevice( dx->fdo);
    }
}
#pragma code_seg() // end PAGE section
```

Заголовочный файл driver.h содержит объявления, необходимые для компиляции драйвера Example.sys. Третий параметр CTL_CODE называется Function и при составлении собственных (пользовательских) IOCTL кодов его значение не должно быть менее 0x800. Пересечение пользовательских IOCTL кодов со значениями IOCTL кодов других драйверов не имеет никакого значения, поскольку они действуют только в пределах конкретного драйвера.

Листинг 2 – Пример простого драйвера (файл driver.h)

```
#ifndef _DRIVER_H_04802_BASHBD_1UIWQ1_8239_1NJKDH832_901_
#define _DRIVER_H_04802_BASHBD_1UIWQ1_8239_1NJKDH832_901_
// Выше приведены две строки (в конце файла имеется еще #endif),
// которые в больших проектах запрещают повторные  проходы по тексту,
// который находится внутри h-файла (что весьма удобно для повышения
// скорости компиляции).
// (Файл Driver.h)

#ifdef __cplusplus
extern "C"
{
#endif

#include "ntddk.h"

//#include "wdm.h"
// ^^^^^^^^^^^^^^ если выбрать эту строку и закомментировать
// предыдущую, то компиляция  в среде DDK (при помощи утилиты Build)
// также пройдет успешно, однако драйвер Example не станет от этого
// настоящим WDM драйвером.

#ifdef __cplusplus
}
#endif
// Определяем структуру расширения устройства. Включим в нее
// указатель на FDO (для удобства последующей работы UnloadRoutine) и
// имя символьной ссылки в формате UNOCODE_STRING.

typedef struct _EXAMPLE_DEVICE_EXTENSION
{
```

```
   PDEVICE_OBJECT  fdo;
   UNICODE_STRING  ustrSymLinkName; // L"\\DosDevices\\Example"
} EXAMPLE_DEVICE_EXTENSION, *PEXAMPLE_DEVICE_EXTENSION;

// Определяем собственные коды IOCTL, с которыми можно будет
// обращаться к драйверу при помощи вызова DeviceIoControl.
// Определение макроса CTL_CODE можно найти в файле DDK Winioctl.h.
// Там же можно найти и численные значения, скрывающиеся под именами
// METHOD_BUFFERED и METHOD_NEITHER.

// Внимание! Текст, приведенный ниже, должен войти в файл Ioctl.h,
// который будет необходим для компиляции тестового приложения.
// (Разумеется, за исключением последней строки с "#endif".)

#define IOCTL_PRINT_DEBUG_MESS CTL_CODE( \
   FILE_DEVICE_UNKNOWN, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_CHANGE_IRQL CTL_CODE(\
   FILE_DEVICE_UNKNOWN, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_MAKE_SYSTEM_CRASH CTL_CODE( \
   FILE_DEVICE_UNKNOWN, 0x803, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_TOUCH_PORT_378H CTL_CODE( \
   FILE_DEVICE_UNKNOWN, 0x804, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_SEND_BYTE_TO_USER CTL_CODE( \
   FILE_DEVICE_UNKNOWN, 0x805, METHOD_BUFFERED, FILE_ANY_ACCESS)

// Вариант :
//#define IOCTL_SEND_BYTE_TO_USER CTL_CODE( \
//    FILE_DEVICE_UNKNOWN, 0x805, METHOD_NEITHER, FILE_ANY_ACCESS)
#endif
```

5.2.2 Компиляция и сборка драйвера

Для компиляции и сборки драйвера утилитой Build пакета DDK потребуется создать два файла описания проекта – Makefile и Sources.

Листинг 3 – Файл Makefile

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

!INCLUDE $(NTMAKEENV)\makefile.def
```

Этот файл управляет работой программы Build и в нашем случае имеет стандартный вид (его можно найти практически в любой директории примеров DDK).

Файл sources отражает индивидуальные настройки процесса компиляции и сборки.

Листинг 4 – Файл Sources

```
# Файл Sources
TARGETNAME=Example
TARGETTYPE=DRIVER
#DRIVERTYPE=WDM
TARGETPATH=obj
SOURCES=init.cpp
```

Данный файл задает имя выходного файла Example. Поскольку проект (TARGETTYPE) имеет тип DRIVER, то выходной файл будет иметь расширение .sys. Промежуточные файлы будут размещены во вложенной директории .\obj. Строка SOURCES задает единственный файл с исходным текстом — это файл init.cpp.

Если бы мы выполняли компиляцию и сборку WDM драйвера, то нужно было бы в тексте Driver.h использовать #include "wdm.h" (взять определения из заголовочного файла "wdm.h" вместо "ntddk.h"), а в данном файле Sources — удалить символ '#' (который вводит строку-комментарий) в первой позиции третьей строки. После этого строка

DRIVERTYPE=WDM

стала бы указывать утилите Build на то, что выполняется компиляция и сборка WDM драйвера.

Разместим для определенности все файлы (нам понадобятся файлы init.cpp, Driver.h, Makefile и sources) в директорию C:\Example.

Для компиляции следует выбрать в меню запуска программ «Пуск => Программы => ... запуск соответствующей среды», в результате чего появится консольное окно, для которого уже (автоматически) будут должным образом установлены переменные окружения. Если у разработчика имеются файлы makefile, sources (описывающие процесс сборки данного конкретного драйвера), и пакет DDK установлен корректно, то необходимо лишь перейти в рабочую директорию проекта командой cd (для нашего драйвера это директория C:\Example) и ввести команду build. В случае ошибок компиляции или сборки вывод будет содержать и их диагностику.

Результат сборки можно будет найти в поддиректории .\objchk_w2k\i386 (поскольку подразумеваются настройки переменных среды сборки под Windows 2000).

5.2.3 Инсталляция и запуск драйвера

Существует несколько способов инсталляции и запуска данного драйвера. Следует отметить, что в других случаях выбор может быть не столь разнообразен, в частности, WDM драйверы рекомендуется инсталлировать при помощи Мастера Установки нового оборудования и inf файла.

5.2.3.1 Инсталляция внесением записей в Системный Реестр

Метод инсталляции путем внесения записей в Системный Реестр (Registry) не требует никаких вспомогательных программных средств – только редактор Системного Реестра. Редактирование Системного Реестра, предлагаемое ниже, не является критичным для работоспособности операционной системы.

Для запуска драйвера следует переписать его бинарный файл Example.sys в директорию C:\Windows\System32\Drivers и создать файл (назовем его ExampleNT.reg).

Файлы импорта в Системный Реестр Windows NT должны быть в формате UNICODE, для этого можно воспользоваться стандартным приложением «Блокнот» (Notepad), но при сохранении (выбрав пункт «Сохранить как…» (Save As…)) явным образом указать кодировку Unicode.

Листинг 5 – Файл ExampleNT.reg

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Example]
"ErrorControl"=dword:00000001
"Type"=dword:00000001
"Start"=dword:00000002
"ImagePath"="\\SystemRoot\\System32\\Drivers\\Example.sys"
```

После этого следует войти в редактор Системного Реестра (Пуск => Выполнить => regedit) и произвести импорт созданного файла. Импорт данного файла в Реестр можно выполнить, если дважды кликнуть мышкой на этом файле в стандартной программе Проводник.

В результате импорта в Системном Реестре будет создан новый подраздел \Example в ветви HKLM\System\CurrentControlSet\Services. В этот подраздел будут занесены параметры ErrorControl, ImagePath, Start и Type.

Параметр Type определяет драйвер режима ядра (значение 1).

Параметр ImagePath определяет местонахождение файла загружаемого модуля (в нашем случае - C:\Windows\System32\Drivers\Example.sys).

Параметр Start определяет момент загрузки сервиса – автостарт после загрузки системы (значение 2).

Параметр ErrorControl определяет поведение системы при возникновении ошибок во время загрузки данного модуля. В данном случае (значение 1) означает следующее: в процессе загрузки ошибки игнорируются, но выводятся сообщения о них, при этом загрузка продолжается.

Несмотря на простоту внесения изменений в Системный Реестр путем импорта заранее созданного текстового файла соответствующего формата, этот метод следует применять с большой осторожностью, поскольку новая информация легко переписывает предшествующую. Возможно, следует дополнительно побеспокоиться о сохранении прежних данных, которые подвергаются модификации (например, путем предварительного экспорта модифицируемых разделов в файл на диске средствами штатного редактора Системного Реестра).

После того как выполнена описанная модификация Системного Реестра и файл Example.sys был размещен в директории C:\Windows\System32\Drivers\, необходимо выполнить перезагрузку операционной системы для того, чтобы драйвер был загружен и начал работу.

5.2.3.2 Инсталляция с использованием INF файла

Для такого способа инсталляции драйвера потребуется создать текстовый файл (назовем его Example.inf), в котором будет представлена информация для работы Мастера Установки нового оборудования. В данном файле имеет значение даже то, куда поставлена запятая, поэтому его следует повторить в точности.

Листинг 6 – Файл Example.inf

```
; Example.Inf - install information file
;  Created 2 feb 2003 by SVP
[Version]
Signature="$Chicago$"
Class=Unknown
Provider=%SVPBook%
DriverVer=09/01/2008,1.0.0.2

[Manufacturer]
%SVPBook%=SVP.Science

[SVP.Science]
%Example%=Example.Install, *svpBook\Example

[DestinationDirs]
Example.Files.Driver=10,System32\Drivers  ; куда копировать для Win98
Example.Files.Driver.NTx86=10,System32\Drivers ; куда копировать для NT

[SourceDisksNames]
1="Example build directory",,,       ; первая цифра - единица
```

```
[SourceDisksFiles]
Example.sys=1,drv\w98   ; где находится новый драйвер для Win98

[SourceDisksFiles.x86]
Example.sys=1,drv\nt   ; где находится новый драйвер для NT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Windows 98
[Example.Install]
CopyFiles=Example.Files.Driver
AddReg=Example.AddReg

[Example.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,Example.sys
[Example.Files.Driver]
Example.sys
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Windows 2000, XP, Server 2003
[Example.Install.NTx86]
CopyFiles=Example.Files.Driver.NTx86

[Example.Files.Driver.NTx86]
Example.sys,,,%COPYFLG_NOSKIP%

[Example.Install.NTx86.Services]
AddService = Example, %SPSVCINST_ASSOCSERVICE%, Example.Service

[Example.Service]
DisplayName    = %Example.ServiceName%
ServiceType    = %SERVICE_KERNEL_DRIVER%
StartType      = %SERVICE_AUTO_START%
ErrorControl   = %SERVICE_ERROR_NORMAL%
ServiceBinary  = %10%\System32\Drivers\Example.sys
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Strings
[Strings]
SVPBook="Introduction to Driver Programming"
Example="Example driver: checked build"
Example.ServiceName="Example NTDDK driver (V.001)"

SPSVCINST_ASSOCSERVICE=0x00000002
COPYFLG_NOSKIP=2   ; Do not allow user to skip file
SERVICE_KERNEL_DRIVER=1
SERVICE_AUTO_START=2
SERVICE_DEMAND_START=3
SERVICE_ERROR_NORMAL=1
```

Для проведения инсталляции не рекомендуется пользоваться директориями, имеющими в названии пробелы и символы кириллицы (например, "C:\Пример драйвера\").

В каталог инсталляции следует поместить файл Example.inf, а также создать директорию drv со вложенными поддиректориями drv\w98 и drv\nt, куда следует поместить по одной копии файла драйвера Example.sys.

Теперь можно приступать к установке драйвера при помощи Мастера Установки нового оборудования (Пуск => Настройка => ...). При его работе важно выполнить следующие действия:

1. Следует самостоятельно выбрать устанавливаемое устройство (а не пользоваться услугами автоматического обнаружения).

2. Выбрать установку драйвера с диска, после чего следует указать директорию на жестком диске, где находится Example.inf, поддиректории \drv\nt и \drv\w98 и две копии Example.sys, как было указано выше.

После идентификации inf файла Мастер Установки нового оборудования самостоятельно скопирует файл Example.sys из соответствующей директории drv\w98 или drv\nt (в нашем случае эти файлы идентичны) в \System32\Drivers внутри системной директории. Мастер Установки произведет модификацию записей в Системном Реестре, в результате чего драйвер будет загружаться после загрузки системы (когда она произойдет в следующий раз).

Для запуска данного драйвера сразу после установки Мастером Установки не требуется перезагрузки системы.

По завершении работы Мастера Установки драйвер готов к использованию и обращению к нему из консольного приложения, описанного ниже. Результаты работы Мастера Установки с записями Системного Реестра следует искать в разделе HKLM\System\CurrentControlSet\Services\Unknown\Example (для Windows 2000/XP/Server 2003).

Следует отметить, что информацию о драйвере Example.sys после установки можно увидеть в Настройках Системы (Система/Диспетчер Устройств в Windows NT), однако многие информационные поля там не будут определены (в случае Windows NT таких полей будет меньше). Это объясняется тем, что информация, для которой указано "неизвестна" должна поступать из файла драйвера, для чего в нем должны быть предусмотрены информационные ресурсы, обычно размещающиеся в .rc файле проекта. В данном проекте такого файла нет, поэтому не вся желаемая информация предоставляется системным службам.

Другой вопрос, который может возникнуть после описанной процедуры: почему мы смогли установить драйвер, в сущности, "никакого" устройства?! Ответ также несложен. Поскольку к системе могут подключаться устройства, не поддерживающие PnP (legacy devices), которые не могут быть автоматически обнаружены и которые не могут быть подключены (загружены их драйверы) иначе, чем по указанию администратора

системы, то фирма Microsoft обязана предоставить способ установки драйверов "по желанию". Что и произошло в нашем случае.

5.2.3.3 Приложение для тестирования драйвера Example.sys

Перед тем, как приступить к тестированию драйвера путем вызова его сервисов из приложения, следует это приложение создать. И хотя драйвер можно успешно запускать программой Monitor, воспользуемся функциями SCM.

Листинг 7 – Файл ExampleTest.cpp

```
//////////////////////////////////////////////////////////////////
// (Файл ExampleTest.cpp)
// Консольное приложение для тестирования драйвера Example.sys
//////////////////////////////////////////////////////////////////

// Заголовочные файлы, которые необходимы в данном приложении:
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Внимание! Файл Ioctl.h должен быть получен из файла Driver.h
// (см. комментрарии в Driver.h) и размещен в одной директории с
// данным файлом (TestExam.cpp).
#include "Ioctl.h"

// Имя объекта драйвера и местоположение загружаемого файла
#define DRIVERNAME    _T("Example")
//#define DRIVERBINARY  _T("C:\\Example\\Example.sys")
//#define DRIVERBINARY  _T("C:\\Ex\\objchk_w2k\\i386\\Example.sys")
#define DRIVERBINARY  _T("C:\\Ex\\tester\\Example.sys")

// Функция установки драйвера на основе SCM вызовов
BOOL InstallDriver(SC_HANDLE scm, LPCTSTR DriverName, LPCTSTR driverExec)
{
   SC_HANDLE Service =
           CreateService ( scm,    // открытый дескриптор к SCManager
                   DriverName,      // имя сервиса - Example
                   DriverName,      // для вывода на экран
                   SERVICE_ALL_ACCESS,    // желаемый доступ
                   SERVICE_KERNEL_DRIVER, // тип сервиса
                   SERVICE_DEMAND_START,  // тип запуска
                   SERVICE_ERROR_NORMAL,  // как обрабатывается ошибка
                   driverExec,            // путь к бинарному файлу
                   // Остальные параметры не используются - укажем NULL
                   NULL,    // Не определяем группу загрузки
                   NULL, NULL, NULL, NULL);
   if (Service == NULL) // неудача
   {
       DWORD err = GetLastError();
```

```
        if (err == ERROR_SERVICE_EXISTS) {/* уже установлен */}
        // более серьезная ошибка:
        else  printf ("ERR: CanTt create service. Err=%d\n",err);
        // (^^ Этот код ошибки можно подставить в ErrLook):
        return FALSE;
    }
    CloseServiceHandle (Service);
return TRUE;
}

// Функция удаления драйвера на основе SCM вызовов
BOOL RemoveDriver(SC_HANDLE scm, LPCTSTR DriverName)
{
    SC_HANDLE Service =
            OpenService (scm, DriverName, SERVICE_ALL_ACCESS);
    if (Service == NULL) return FALSE;
    BOOL ret = DeleteService (Service);
    if (!ret) { /* неудача при удалении драйвера */ }

    CloseServiceHandle (Service);
return ret;
}

// Функция запуска драйвера на основе SCM вызовов
BOOL StartDriver(SC_HANDLE  scm, LPCTSTR DriverName)
{
    SC_HANDLE Service =
            OpenService(scm, DriverName, SERVICE_ALL_ACCESS);
    if (Service == NULL) return FALSE; /* open failed */
    BOOL ret =
            StartService( Service, // дескриптор
                          0,       // число аргументов
                          NULL  ); // указатель  на аргументы
    if (!ret) // неудача
    {
        DWORD err = GetLastError();
        if (err == ERROR_SERVICE_ALREADY_RUNNING)
            ret = TRUE; // OK, драйвер уже работает!
        else { /* другие проблемы */}
    }

    CloseServiceHandle (Service);
return ret;
}
// Функция останова драйвера на основе SCM вызовов
BOOL StopDriver(SC_HANDLE  scm, LPCTSTR DriverName)
{
    SC_HANDLE Service =
            OpenService (scm, DriverName, SERVICE_ALL_ACCESS );
    if (Service == NULL)  // Невозможно выполнить останов драйвера
    {
        DWORD err = GetLastError();
```

```
        return FALSE;
    }
    SERVICE_STATUS serviceStatus;
    BOOL ret =
    ControlService(Service, SERVICE_CONTROL_STOP, &serviceStatus);
    if (!ret)
    {
        DWORD err = GetLastError();
        // дополнительная диагностика
    }

    CloseServiceHandle (Service);
return ret;
}

// Соберем вместе действия по установке, запуску, останову
// и удалению драйвера.
// (Однако пользоваться этой функцией в данном примере не придется.)
/* Закомментируем ее.
void Test_SCM_Installation(void)
{
    SC_HANDLE scm = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
    if(scm == NULL) // неудача
    {
        // Получаем код ошибки и ее текстовый эквивалент
        unsigned long err = GetLastError();
        PrintErrorMessage(err);
        return;
    }
    BOOL res;
    res = InstallDriver(scm, DRIVERNAME, DRIVERBINARY );
    // Ошибка может оказаться не фатальной. Продолжаем:
    res = StartDriver  (scm, DRIVERNAME );
    if(res)
    {
        //E Здесь следует разместить функции работы с драйвером
        .. .. ..
        res = StopDriver   (scm, DRIVERNAME );
        if(res) res = RemoveDriver (scm, DRIVERNAME );
    }
CloseServiceHandle(scm);
return;
}
*/

#define SCM_SERVICE
// ^^^^^^^^^^^^^^^^ вводим элемент условной компиляции, при помощи
// которого можно отключать использование SCM установки драйвера
// в тексте данного приложения. (Здесь использование SCM включено.)

// Основная функция тестирующего приложения.
// Здесь минимум внимания уделен диагностике ошибочных ситуаций.
```

```
// В действительно рабочих приложениях следует уделить этому
// больше внимания!

int __cdecl main(int argc, char* argv[])
{
   #ifdef SCM_SERVICE
   // Используем сервис SCM для запуска драйвера.
   BOOL res; // Получаем доступ к SCM :
   SC_HANDLE scm = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
   if(scm == NULL) return -1; // неудача

   // Делаем попытку установки драйвера
   res = InstallDriver(scm, DRIVERNAME, DRIVERBINARY );
   if(!res) // Неудача, но возможно, он уже инсталлирован
       printf("Cannot install service");

   res = StartDriver  (scm, DRIVERNAME );
   if(!res)
   {
       printf("Cannot start driver!");
       res = RemoveDriver (scm, DRIVERNAME );
       if(!res)
       {
           printf("Cannot remove driver!");
       }
       CloseServiceHandle(scm); // Отключаемся от SCM
       return -1;
   }
   #endif

   HANDLE hHandle =           // Получаем доступ к драйверу
           CreateFile( "\\\\.\\Example",
                       GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL );
   if(hHandle==INVALID_HANDLE_VALUE)
   {
       printf("ERR: can not access driver Example.sys!\n");
       return (-1);
   }
   DWORD BytesReturned;    // Переменная для хранения числа
                           // переданных байт
   // Последовательно выполняем обращения к драйверу
   // с различными кодами IOCTL:

   unsigned long ioctlCode=IOCTL_PRINT_DEBUG_MESS;
   if( !DeviceIoControl(   hHandle,
                   ioctlCode,
                   NULL, 0,    // Input
```

```
                NULL, 0,    // Output
                &BytesReturned,
                NULL )  )
{
    printf( "Error in IOCTL_PRINT_DEBUG_MESS!" );
    return(-1);
}

ioctlCode=IOCTL_CHANGE_IRQL;
if( !DeviceIoControl(   hHandle,
                ioctlCode,
                NULL, 0,    // Input
                NULL, 0,    // Output
                &BytesReturned,
                NULL )  )
{
    printf( "Error in IOCTL_CHANGE_IRQL!" );
    return(-1);
}

ioctlCode=IOCTL_TOUCH_PORT_378H;
if( !DeviceIoControl(   hHandle,
                ioctlCode,
                NULL, 0,  // Input
                NULL, 0,  // Output
                &BytesReturned,
                NULL )  )
{
    printf( "Error in IOCTL_TOUCH_PORT_378H!" );
    return(-1);
}

// Следующий тест. Получаем 1 байт данных из драйвера.
// По окончании данного вызова переменная xdata должна
// содержать значение 33:
unsigned char xdata = 0x88;
ioctlCode=IOCTL_SEND_BYTE_TO_USER;
if( !DeviceIoControl(   hHandle,
                ioctlCode,
                NULL, 0,  // Input
                &xdata, sizeof(xdata),// Output
                &BytesReturned,
                NULL )  )
{
    printf( "Error in IOCTL_SEND_BYTE_TO_USER!" );
    return(-1);
}

// Вывод диагностического сообщения в консольном окне:
printf("IOCTL_SEND_BYTE_TO_USER: BytesReturned=%d xdata=%d",
                BytesReturned, xdata);
```

```
    // Выполнение следующего теста в Windows NT приведет к
    // фатальному сбою операционной системы (намеренно выполненное
    // падение ОС может быть полезно при изучении, например,
    // организации crash dump файла и работы с отладчиком).
    /*
    ioctlCode=IOCTL_MAKE_SYSTEM_CRASH;
    if( !DeviceIoControl(   hHandle,
                    ioctlCode,
                    NULL, 0,                // Input
                    NULL, 0,    // Output
                    &BytesReturned,
                    NULL )  )
    {
        printf( "Error in IOCTL_MAKE_SYSTEM_CRASH!" );
        return(-1);
    }
    */
    // Закрываем дескриптор доступа к драйверу:
    CloseHandle(hHandle);


    #ifdef SCM_SERVICE
    // Останавливаем и удаляем драйвер. Отключаемся от SCM.
    res = StopDriver   (scm, DRIVERNAME );
    if(!res)
    {
        printf("Cannot stop driver!");
        CloseServiceHandle(scm);
        return -1;

    }

    res = RemoveDriver (scm, DRIVERNAME );
    if(!res)
    {
        printf("Cannot remove driver!");
        CloseServiceHandle(scm);
        return -1;
    }

    CloseServiceHandle(scm);
    #endif

return 0;
}
```

Как уже было сказано, из всех возможных способов инсталляции и запуска драйвера Example.sys, ниже будет использован способ тестирования с применением тестирующего консольного приложения, которое само будет выполнять инсталляцию и удаление драйвера (прибегая к вызовам SCM Менеджера). Для поэтапного ознакомления с процессом взаимодействия драйвера и обращающегося к нему приложения рекомендуется запу-

стить программу ExampleTest под отладчиком (например, Visual Studio) в пошаговом режиме.

Перед запуском тестирующей программы ExampleTest рекомендуется загрузить программу DebugView, чтобы в ее рабочем окне наблюдать сообщения, поступающие непосредственно из кода драйвера Example.sys (отладочной сборки).

Однако прежде чем перейти к рассмотрению сообщений от драйвера, после выполнения установки и запуска драйвера, прежде следует обратиться к программам WinObj и DeviceTree или аналогичным им программным средствам для того, чтобы удостовериться, присутствует ли в них информация об установленном драйвере.

Как уже было сказано протокол полученных программой DebugView отладочных сообщений драйвера можно сохранить в файле для последующего анализа.

5.2.4 Структура драйвера Windows NT (WDM Driver)

Проблемы конфигурирования не давали покоя специалистам и раньше, но распространенность вычислительной техники сделали преодоление этой проблемы делом почти что первостепенным.

Решение пришло в виде разработки спецификации Plug and Play, согласно которой устройства должны выдерживать определенные механические и электрические нормы. Основное же требование Plug and Play состоит в том, что устройства должны уметь предоставлять идентификационную информацию о себе в формате, определенном для данного типа (PCI, USB, FireWire, CardBus) подключения.

Главным плюсом использования методологии Plug and Play является обеспечение автоматической поддержки инсталляции и удаления системных устройств.

Поддержка PnP в операционной системе обеспечивается следующими компонентами:

1. **PnP Менеджер**, который состоит из двух частей – работающей в режиме ядра и работающей в пользовательском режиме. Часть из режима ядра взаимодействует с аппаратурой и другими программными компонентами, функционирующими в режиме ядра, обеспечивая управление правильным определением и конфигурированием аппаратуры. Часть из пользовательского режима взаимодействует с компонентами пользовательского интерфейса, позволяя интерактивной программе делать запросы и изменять конфигурацию инсталлированного PnP программного обеспечения.

2. **Менеджер Управления Энергопитанием** (Power Manager), который определяет и обрабатывает события энергообеспечения.

3. **Системный Реестр Windows**, являющийся базой данных установленного аппаратного и программного обеспечения, поддерживающего

спецификацию PnP. Содержимое реестра помогает драйверам и другим компонентам при определении ресурсов, используемых любым конкретным устройством.

4. **Inf-файлы**. Каждое устройство должно быть полностью описано файлом, который используется при инсталляции управляющего им драйвера. Inf-файл является рецептом, как и какую информацию об устройстве заносить, в частности, в Системный Реестр.

5. **Драйверы для PnP устройств**, которые можно разделить на две категории: WDM и NT драйвера.

Драйверы для PnP устройств можно разделить на две категории: WDM и NT драйвера. Последние являются "унаследованными" от NT драйверами (Legacy Drivers), которые опираются на некоторые аспекты PnP архитектуры, но, с другой стороны, не полностью удовлетворяют модели WDM. Например, они могут использовать сервисы PnP Менеджера, чтобы получить информацию о конфигурации, но при этом не обрабатывают IRP пакеты сообщения с кодом IRP_MJ_PNP. Драйверы WDM модели, по определению, полностью соответствуют требованиям взаимодействия по правилам PnP.

В драйверах модели WDM функция DriverEntry все еще служит в качестве начальной точки соприкосновения операционной системы с драйвером. Однако теперь обязанности ее сократились. В частности, роль DriverEntry теперь состоит только в том, чтобы "опубликовать" (передать Диспетчеру ввода/вывода соответствующие адреса для вызова функций по адресу) остальные процедуры драйвера. Теперь DriverEntry не создает объект устройства (Device Object) для подконтрольного аппаратного обеспечения.

Обязанности по созданию объекта устройства возложены на новую функцию драйвера AddDevice, которая теперь публикуется (обратите внимание!) в структуре расширения драйвера (Driver Extension) во время работы DriverEntry. Структура расширения драйвера является строго определенной структурой – не следует путать ее с определяемой разработчиком драйвера структурой расширения устройства (Device Extension). Пример публикации AddDevice был закомментирован в теле функции DriverEntry для Legacy Driver’a. Повторим ее еще раз:

```
DriverObject->DriverExtension->AddDevice = MyAddDeviceRoutine;
```

Основной обязанностью MyAddDeviceRoutine является создание объекта устройства (теперь уже – функционального) с использованием вызова системного **IoCreateDevice** и, скорее всего, подключение его к объекту физического устройства (вызовом **IoAttachDevice**), указатель на который поступает в параметре pPDO.

Драйверная модель WDM ориентирована на многослойные драйверные структуры и базируется на подходе, в котором функциональные драйверы позиционируются над физическими драйверами, возможно, при участии фильтр-драйверов, окружающих функциональный слой. Во многих случаях, требуется только лишь создать фильтр-драйвер чтобы добиться нужного результата от существующего функционального драйвера.
В рамках многослойного подхода можно определить три типа драйверов:

1. **Шинные драйверы** – обеспечивают интерфейс аппаратных шин в базисе "один слот – одна единица" и создают один или более физических объектов устройств (PDO, Physical Device Object), соответствующих каждому обнаруженному устройству, подключенному к шине. Шинный драйвер конструирует PDO и управляет им, вследствие чего часто его называют физическим драйвером.

2. **Функциональные драйверы** – обеспечивают чтение, запись и прочую логику функционирования отдельного устройства. Они создают и управляют одним или более функциональными объектами устройств (FDO, Functional Device Object).

3. **Фильтр-драйверы** – обеспечивают модификацию запроса на ввод/вывод перед предъявлением его драйверам более низких уровней. Фильтры могут быть размещены вокруг функционального драйвера либо над шинным драйвером.

Драйверная модель WDM построена на организации и манипуляции слоями Объектов Физических устройств (Physical Device Object, PDO) и Объектов Функциональных устройств (Functional Device Object, FDO). Объект PDO создается для каждого физически идентифицируемого элемента аппаратуры, подключенного к шине данных, и подразумевает ответственность за низкоуровневый контроль, достаточно общий для набора функций, реализуемых этим аппаратным элементом. Объект FDO предлагает "олицетворение" каждой логической функции, которую "видит" в устройстве программное обеспечение верхних уровней.

В качестве примера рассмотрим привод жесткого диска и его драйвер. Привод диска может быть представлен объектом PDO, который реализует функции шинного адаптера (присоединяет IDE диск к шине PCI). Как только возникает PDO объект, можно реализовывать объект FDO, который примет на себя выполнение функциональных операций над собственно диском. Обращаясь к FDO, можно будет сделать конкретный функциональный запрос к диску, например, чтение или запись сектора. Однако FDO может выбрать и передачу без модификации конкретного запроса своим партнерам по обслуживанию данного устройства (например, сообщение о снижении напряжения питания).

В действительности, роль PDO объектов быстро усложняется и становится рекурсивной. Например, USB хост-контроллер начинает жизнь как

физическое устройство, подключенное к шине PCI. Но вскоре этот хост-контроллер сам начинает выступать в роли шинного драйвера и, по мере обнаружения устройств, подключенных к USB шине, создает свою коллекцию PDO объектов, каждый из которых контролирует собственный FDO объект.

Эта методология в дальнейшем усложняется еще более, поскольку Функциональным Объектам устройств (FDO) разрешается окружать себя Объектами-Фильтрами (filter device objects, FiDO). Соответственно, каждому FiDO объекту сопоставлен драйвер, выполняющий определенную работу. Эти фильтрующие объекты верхнего и нижнего уровня могут существовать в любом количестве. Назначение их в том, чтобы модифицировать или обогатить процесс обработки запросов ввода/вывода возможностью использования всего результирующего стека объектов устройств. Следует отметить, что FDO и FiDO объекты отличаются только в смысловом отношении — FDO объект и его драйвер являются главной персоной, FiDO объекты и их драйверы являются вспомогательными (вплоть до того, что предпочитают не иметь собственных имен).

Для того чтобы сделать различие между FDO объектами, которые представляют аппаратные шины, и FDO объектами, которые аппаратные шины не представляют, в документации DDK используются термины ***шинные FDO (bus FDO)*** и ***не-шинные FDO (nonbus FDO)***. Первые реализуют обязанности драйвера по перечислению (enumerating) всех устройств, подключенных к шине. Такой шинный FDO объект затем создает новые PDO объекты для каждого из подключенных к шине устройств.

Добавляет проблем тот факт, что существует лишь небольшая смысловая разница между не-шинным FDO и фильтрующим объектом устройства (filter device object). С точки зрения Менеджера PnP, все объекты устройств позиционируют себя в стеке устройств (device stack), а тот факт, что некоторые устройства считают себя более чем просто объектами-фильтрами, кажется ему малозначительным.

Последовательность в стеке устройств показана на рисунке 1. Различия между шинными и не-шинными FDO отражены на рисунке 2.

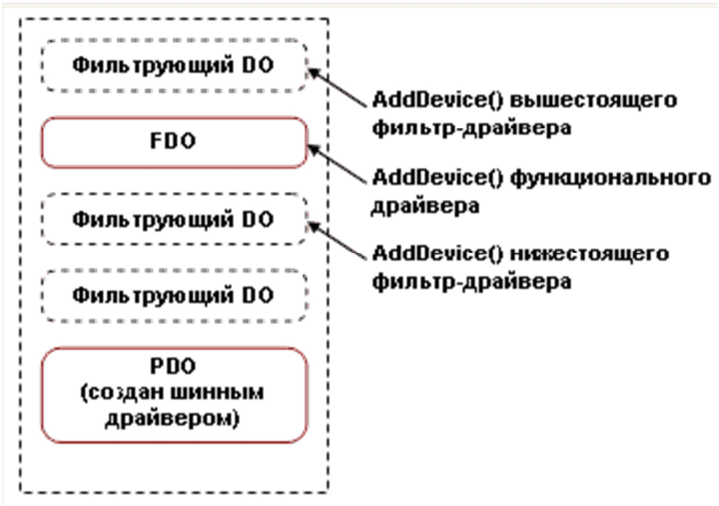

Рисунок 1 – Стек устройств

Рисунок 2 – Шинные и не-шинные FDO

Понимание концепции стека устройств важно для того, чтобы правильно описать, когда вызывается функция AddDevice конкретного устройства. Общий алгоритм, используемый для загрузки драйверов и вовлечения в работу функции MyAddDeviceRoutine, описывается следующей последовательностью:

1) Во время инсталляции операционной системы, операционная система обнаруживает и составляет список (enumerate) всех шин в Системном Реестре (System Registry). Кроме того, детектируется и регистрируется топология и межсоединения этих шин.
2) Во время процесса загрузки производится загрузка шинного драйвера для каждой известной системе шины. Как правило, Microsoft поставляет все шинные драйверы, однако могут быть установлены и специализированные драйвера для патентованных шин данных.
3) Одна из первоочередных задач шинного драйвера состоит в том, чтобы составить перечень (enumerate) всех устройств, подклю-

ченных к шине. Объект PDO создается для каждого обнаруженного устройства.

4) Для каждого обнаруженного устройства в Системном Реестре определен класс устройств (class of device), который определяет верхний и нижний фильтры, если таковые имеются, так же, как и драйвер для FDO.
5) В случае если фильтрующий драйвер или FDO драйвер еще не загружены, система выполняет загрузку и вызывает DriverEntry.
6) Функция AddDevice вызывается для каждого FDO, которая, в свою очередь, вызывает IoCreateDevice и IoAttachDeviceToDeviceStack, обеспечивая построение стека устройств (device stack).

Функция IoAttachDeviceToStack вызывается из AddDevice для того, чтобы разместить FDO в вершине (на текущий момент) стека устройств. Прототип функции **IoAttachDeviceToStack:**

```
//Выполняет подключение вновь созданного объекта устройства,
//pNewDevice, к стеку устройств
PDEVICE_OBJECT IoAttachDeviceToDeviceStack(
//Указатель на подключаемый к стеку объект (созданный в данном драйвере)
IN PDEVICE_OBJECT pNewDevice,
//Указатель на объект устройства,
//к которому подключается новое устройство
IN PDEVICE_OBJECT pOldDevice);
```

Возвращаемое значение: указатель на устройство, бывшее на вершине стека до данного вызова или NULL (в случае ошибки, например, если драйвер целевого устройства еще не загружен).

Возвращаемый вызовом **IoAttachDeviceToDeviceStack** указатель может отличаться от переданного значения pOldDevice, например, по той причине, что над объектом устройства pOldDevice уже размещены объекты устройств, предоставленные фильтр-драйверами.

Для подключения данного объекта устройства (по указателю pNewDevice) необходимо владеть указателем на целевой объект устройства (pOldDevice)

Хорошо, когда драйвер подключает свой объект устройства к родительскому объекту устройства (шинного драйвера), указатель на который поступает в процедуру AddDevice при вызове через заголовок. Но если необходимо подключить новый объект устройства к объекту устройства другого драйвера, отличающегося от pPDO, то имеется два пути.

При подключении драйвера к произвольному объекту устройства можно поступить двумя способами. Во-первых, если известно имя нужного устройства, можно получить указатель на искомый объект устройства, воспользовавшись предварительно вызовом **IoGetDeviceObjectPointer**.

Полученный указатель на искомый объект устройства (возвращаемый по адресу ppDevObj), можно применить в вызове **IoAttachDeviceToDeviceStack**, описанном выше.

```
// Получает указатель на объект устройства по имени устройства
NTSTATUS IoGetDeviceObjectPointer(
// Имя устройств
IN PUNICODE_STRING DeviceName,
// Маска доступа: FILE_READ_DATA, FILE_WRITE_DATA или FILE_ALL_ACCESS
IN ACCESS_MASK Access,
// Указатель на файловый объект, которым представлен искомый объект
// устройства для кода пользовательского режима
OUT PFILE_OBJECT *ppFileObj,
// Указатель на искомый объект устройства
OUT PDEVICE_OBJECT *ppDevObj);
```

Возвращаемое значение STATUS_SUCCESS или STATUS_Xxx — код ошибки.

Вызов **IoGetDeviceObjectPointer** может быть интересен и тем, что драйвер мог бы адресовать IRP запросы непосредственно искомому объекту устройства (при помощи **IoCallDriver**), создавая IRP пакеты с размером стека StackSize+1, где значение StackSize получено из найденного объекта устройства.

Второй способ подключения к стеку устройств через объект устройства с известным именем осуществляется при помощи вызова **IoAttachDevice**, прототип которого представлен ниже.

```
//Выполняет подключение вновь созданного объекта устройства, pNewDevice
NTSTATUS IoAttachDevice(
// Указатель на подключаемый объект устройства
IN PDEVICE_OBJECT pNewDevice,
// Имя целевого устройства
IN PUNICODE_STRING TagDevName,
// Указатель на объект устройства, к которому подключается
// новое устройство (точнее, указатель на место для указателя)
OUT PDEVICE_OBJECT *ppTagDevice);
```

Возвращаемое значение STATUS_SUCCESS или STATUS_Xxx — код ошибки

В результате вызовов **IoAttachDeviceToDeviceStack** или **IoAttachDevice** будет найден объект устройства, находящийся на вершине стека над указанным целевым объектом (по имени или по указателю). К нему и будет подключен новый объект устройства. Соответственно, разработчик, подключающий свой объект устройства к устройству в "середине"

стека и надеющийся, что таким образом через его драйвер будут "протекать" IRP запросы от вышестоящих драйверов к нижестоящим, глубоко заблуждается. На самом деле, для достижения этой цели необходимо не просто выполнить подключение к нужному объекту устройства, но и сделать это в строго определенный момент загрузки — ранее, чем будет выполнена загрузка вышестоящих драйверов, чьи запросы предполагается перехватывать.

Полученный указатель на объект устройства, к которому произведено подключение, следует сохранить, поскольку он может понадобиться, например, в обработчике запросов IRP_MJ_PNP, см. ниже. Это можно сделать в структуре расширения объекта устройства.

Заключительной задачей функции **AddDevice** драйверов модели WDM является создание символьного имени-ссылки (symbolic link name), если это необходимо, для вновь созданных и доступных устройств. Для этого используется вызов **IoCreateSymbolicLink**, применение которого было продемонстрировано ранее в DriverEntry для Legacy Driver'а.

5.2.4.1 Новые рабочие процедуры в WDM драйверах

Процедура AddDevice, вызываемая PnP Менеджером, только лишь производит инициализацию объекта устройства и, если необходимо, структуры данных расширения объекта устройства. В процедуре AddDevice, по правилам хорошего тона WDM модели, действия над собственно аппаратурой не должны совершаться. Но тогда остаются нерешенными две важные задачи: резервирование и конфигурирование аппаратных ресурсов обслуживаемого физического устройства, и инициализация с подготовкой аппаратной части к использованию.

Все это должен сделать драйвер по получении IRP пакета с кодом IRP_MJ_PNP. Такие IRP пакеты посылается PnP Менеджером, когда происходят события включения или выключения устройства, либо возникают вопросы по конфигурированию устройства.

Категория IRP_MJ_PNP пакетов включает запросы широкого спектра, которые детализируются суб-кодами IRP_MN_Xxx. Поскольку они пропускается через единственную рабочую процедуру, то ее обязанностью является вторичная диспетчеризация по этим суб-кодам, содержащимся в IRP пакете и описывающим специфические действия, осуществления которых ожидает PnP Менеджер.

Регистрация новой для WDM модели рабочей процедуры, которой будет поручено обрабатывать запросы IRP_MJ_PNP со всеми подтипами IRP_MN_Xxx, производится традиционным образом в процедуре DriverEntry:

```
pDriverObj->MajorFunction[IRP_MJ_PNP] = MyPnP_Handler;
```

Пример программного кода для осуществления вторичной диспетчеризации на основе суб-кодов IRP_MN_Xxx приводится ниже.

```
NTSTATUS MyPnP_Handler ( IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp )
{
   // Получить указатель на текущую ячейку стека IRP пакета
   PIO_STACK_LOCATION pIrpStackLocation =
      IoGetCurrentIrpStackLocation( pIrp );

   switch (pIrpStackLocation ->MinorFunction) {
   case IRP_MN_START_DEVICE:
   //. . .
   // Внимание. Все ветви оператора switch должны возвратить
   // результаты обработки
   //. . .
   //
   default:
       // если не поддерживается здесь, то передать запрос вниз:
       IoSkipCurrentIrpStackLocation(pIrp);
   return IoCallDriver(. . ., pIrp);
   }
}
```

Первым параметром вызова **IoCallDriver**, разумеется, является указатель на объект устройства, к которому было произведено подключение в процедуре AddDevice.

```
// Обращается к другому драйверу с запросом, сформулированным в пакете IRP
//(запросы типа IRP_MJ_POWER следует выполнять при помощи вызова
PoCallDriver)
NTSTATUS IoCallDriver(
//Указатель на объект устройства, которому адресован IRP запрос
IN PDEVICE_OBJECT pDevObj,
//Указатель на отправляемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение:

1. STATUS_SUCCESS;
2. STATUS_PENDING – в случае, если пакет требует дополнительной обработки;
3. STATUS_Xxx – в случае ошибки.

Вызов **IoSkipCurrentIrpStackLocation** сообщает Диспетчеру ввода/вывода, что драйвер отказывается от дальнейшего участия в судьбе данного IRP пакета. В том случае, если драйвер желает получить управление над IRP пакетом в момент, когда его обработка нижними слоями драй-

веров будет завершена, то он должен воспользоваться системным вызовом **IoCopyCurrentIrpStackLocationToNext** и зарегистрировать процедуру CompletionRoutine. Она будет вызвана в соответствующий момент.

```
//Изменяет указатель стека IRP так, что нижестоящий драйвер
//будет считать текущую ячейку стека IRP своей
VOID IoSkipCurrentIrpStackLocation(
//Указатель на модифицируемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение void.

```
// Копирует содержимое ячейки стека IRP для текущего драйвера
// в ячейку стека для нижестоящего драйвера
VOID IoCopyCurrentIrpStackLocationToNext(
//Указатель на модифицируемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение void.

Процедура завершения ввода/вывода **CompletionRoutine** есть обратный вызов от Диспетчера ввода/вывода, который позволяет перехватить IRP пакет после того, как низкоуровневый драйвер завершит его обработку. Процедура завершения ввода/вывода регистрируется вызовом **IoSetCompletionRoutine**.

```
//Выполняет регистрацию callback-функции завершения обработки IRP пакета
VOID IoSetCompletionRoutine(
//Указатель на отслеживаемый IRP пакет
IN PIRP pIrp,
//Функция, которая должна получить управление,
// когда обработка IRP будет завершена
IN PIO_COMPLETE_ROUTINE CompletionRoutine,
//Параметр, который получит регистрируемая
//callback функция CompletionRoutine
IN PVOID pContext,
//Вызывать CompletionRoutine в случае успешного завершения обработки дан-
ного IRP пакета
IN BOOLEAN doCallOnSuccess,
//Вызывать CompletionRoutine в случае завершения обработки
//данного IRP с ошибкой
IN BOOLEAN doCallOnError,
//Вызывать CompletionRoutine в случае прерванной обработки
//данного IRP пакета
IN BOOLEAN doCallOnCancel);
```

Возвращаемое значение void.

```
//Перехватывает пакет IRP после завершения работы
//нижнего драйверного слоя
NTSTATUS CompletionRoutine(
//Объект устройства (в составе данного драйвера),
//которому был ранее адресован данный IRP пакет
IN PDEVICE_OBJECT pDevObj,
//Указатель на IRP пакет, обработка которого только что завершена
IN PIRP pIrp,
//Аргумент, указанный в IoSetCompleteRoutine
IN PVOID pContext);
```

Возвращаемое значение STATUS_SUCCESS или STATUS_MORE_PROCESSING_REQUIRED.

Однозначно предсказать, на каком уровне IRQL выполняется процедура завершения, невозможно. В том случае, если нижележащий драйвер, вызывает **IoCompleteRequest** с уровня IRQL равного PASSIVE_LEVEL, то процедура завершения находящегося выше драйвера выполняется на уровне PASSIVE_LEVEL. В случае, если лежащий ниже драйвер завершает обработку IRP пакета на уровне DIPATCH_LEVEL (например, из DPC процедуры), то и процедура завершения лежащего выше драйвера выполняется на уровне DISPATCH_LEVEL.

Выполнение программного кода на уровне DISPATCH_LEVEL ограничивается системными вызовами, которые работают на этом уровне IRQL. Разумеется, следует особо позаботиться, чтобы здесь не производилась работа со страничной памятью.

```
//Вызывается, когда драйвер желает полностью завершить обработку
//данного IRP пакета.
//Обеспечивает вызов процедур завершения всех драйверов,
//имеющихся над данным
VOID IoCompleteRequest(
//Указатель на текущий IRP пакет, обработка которого только что завершена
IN PIRP pIrp,
//Величина, на которую следует изменить приоритет потока,
//выполняющего обработку данного IRP пакета.
//Величина IO_NO_INCREMENT используется,
//если никаких изменений делать не нужно.
IN CCHAR PriorBoost);
```

Возвращаемое значение void.

Чтобы устранить упомянутую выше неоднозначность уровня IRQL работы процедуры завершения, можно прибегнуть к следующей уловке. Предположим, что мы имеем программный код рабочей процедуры. Из-

вестно также, что PnP Менеджер (как, впрочем, и Диспетчер ввода/вывода) всегда выполняет вызов рабочей процедуры драйвера на уровне PASSIVE_LEVEL. Тогда, отправляя пакет IRP нижним слоям драйвера, организуем ожидание (средствами объекта события режима ядра) не выходя из кода данной рабочей процедуры, пока отосланный нижним слоям IRP пакет не возвратится в зарегистрированную функцию CompletionRoutine. Как только это произойдет, объект события кодом функции CompletionRoutine будет переведен в сигнальное состояние, и стадия ожидания в основном потоке завершится. Таким образом, мы получим сигнал о завершении обработки пакета IRP на вполне определенном уровне IRQL, равном именно PASSIVE_LEVEL. Полностью данный метод описывается в примере ниже:

```
// код рабочей процедуры, выполняющийся на уровне PASSIVE_LEVEL
. . . . . .
IoCopyCurrentIrpStackLocationToNext(pIrp);

// Резервируем место под объект события:
KEVENT myEvent;
// Инициализируем его, состояние не сигнальное:
KeInitializeEvent( &myEvent, NotificationEvent, FALSE );
// Регистрируем свою процедуру завершения обработки IRP пакета.
// Указатель на объект myEvent передаем как дополнительный параметр.
IoSetCompletionRoutine( pIrp,
                        MyCompleteRoutine,
                        (PVOID)&myEvent,
                        TRUE, TRUE, TRUE);

// Предположим, что указатель на объект устройства,
// к которому был подключен текущий объект устройства, был ранее
// сохранен в структуре расширения текущего объекта устройства.
PDEVICE_EXTENSION pDeviceExtension = (PDEVICE EXTENSION) pDeviceObject-
>DeviceExtension;
PDEVICE_OBJECT pUnderlyingDevObj = pDeviceExtension->pLowerDevice;

// Отправляем IRP пакет на обработку нижними драйверными слоями
IoCallDriver( pUnderlyingDevObj, pIrp );

// Организуем ожидание, пока не закончится работа на нижних уровнях
KeWaitForSingleObject( &myEvent,
                       Execute,
                       KernelMode,
                       FALSE,
                       NULL);

// Теперь завершаем обработку IRP пакета.
// Его адрес не изменился - pIrp.
// По "возвращении" из "ожидания" уровень IRQL остался прежним для
// данного потока.
```

```
// Поскольку Диспетчер ввода/вывода и PnP Менеджер вызывают
// рабочие процедуры драйвера на уровне PASSIVE_LEVEL, то таким
// он в данном потоке и остался.
. . .
}
NTSTATUS MyCompleteRoutine( IN PDEVICE_OBJECT pDevObj,
                            IN PIRP pIrp,
                            IN PVOID pContextArgument )
{
// Вычисляем указатель на Объект События:
PEVENT pEvent = (PEVENT) pContextArgument;
// Устанавливаем его в сигнальное состояние
KeSetEvent( pEvent, 0, FALSE );    // IRQL <=DISPATCH_LEVEL

// Пакет IRP получен. Завершение работы здесь. Но не окончательно.
return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Рассмотрим подробнее работу вызова **IoCompleteRequest**. Когда некий код некоего драйвера делает этот вызов, программный код **IoCompleteRequest** обращается к ячейкам стека IRP пакета и анализирует, зарегистрировал ли верхний (над текущим) драйвер процедуру завершения CompleteRoutine – это как раз отмечено в стеке IRP пакета. В том случае, если таковой процедуры не обнаруживается, указатель стека поднимается и снова выполняется проверка. Если обнаружена зарегистрированная функция, то она выполняется. В том случае, если вызванная таким образом функция возвращает код завершения, отличный от STATUS_MORE_PROCESSING_REQUIRED, то указатель стека снова поднимается, и действия повторяются. Если в результате вызова получен код завершения STATUS_MORE_PROCESSING_REQUIRED, то управление возвращается инициатору вызова **IoCompleteRequest**.

Когда код **IoCompleteRequest** благополучно достигает в своем рассмотрении вершины стека, то Диспетчер ввода/вывода выполняет действия по освобождению данного IRP пакета (наряду с некоторыми другими операциями).

Отсюда несколько важных следствий.

Во-первых, если драйвер сам создал IRP пакет (подробно рассматривается ниже), то вызов **IoCompleteRequest** означает приказ Диспетчеру ввода/вывода заняться его освобождением – поскольку иных драйверов, процедуры завершения которых можно было бы рассматривать, просто нет.

Во-вторых, если текущий драйвер зарегистрировал свою процедуру завершения и, не вызывая нижних драйверов, сразу выполнил **IoCompleteRequest**, то такая процедура завершения вызвана не будет – код **IoCompleteRequest** ее в рассмотрение просто не примет, переходя сразу к анализу ячеек стека IRP для вышестоящих драйверов.

В-третьих, возможна ситуация, когда после выполнения вызова **IoCompleteRequest** драйвером в середине драйверного стека IRP пакет все еще существует. Не исключено, что он не завершен, поскольку один из верхних драйверов (который, возможно, существует) отказался это сделать в своей процедуре завершения, которую он, возможно, зарегистрировал. Однако текущий драйвер таких допущений делать не должен. Впрочем, как и всех остальных предположений относительно верхних драйверных слоев.

В любом случае, после вызова **IoCompleteRequest** драйвер не имеет права прикасаться к IRP пакету, который передан этому вызову как завершаемый. Кроме того, возврат кода STATUS_MORE_PROCESSING_REQUIRED – это практика зарегистрированных процедур завершения, что является "просьбой" Диспетчеру ввода/вывода возвратиться к данной процедуре завершения позже.

5.2.4.2 Передача PnP IRP пакетов нижним драйверным слоям

Для того чтобы соответствовать драйверной модели WDM, драйвер обязан поддерживать обработку специфичных PnP IRP пакетов, каких конкретно – это определяется конкретным типом объекта устройства – нешинный FDO, шинный FDO и PDO. Во всяком случае, IRP пакеты с приведенными в таблице кодами IRP_MN_Xxx должны поддерживаться драйверами всех типов.

Таблица 2 – Минимальный набор обязательно поддерживаемых IRP пакетов

IRP_MN_Xxx	**Значение**
IRP_MN_START_DEVICE	(Pe) Инициализация устройства с заданными ресурсами
IRP_MN_QUERY_STOP_DEVICE	Осуществима ли остановка устройства для возможного переопределения ресурсов?
IRP_MN_STOP_DEVICE	Остановка устройства с потенциальной возможностью перезапуска или удаления из системы
IRP_MN_CANCEL_STOP_DEVICE	Уведомляет, что предыдущий запрос QUERY_STOP не получит дальнейшего развития
IRP_MN_QUERY_REMOVE_DEVICE	Может ли быть выполнено безопасное удаление устройства в текущий момент?
IRP_MN_REMOVE_DEVICE	Выполнить работу, обратную работе AddDevice
IRP_MN_CANCEL_REMOVE_DEVICE	Уведомляет, что предыдущий запрос QUERY_REMOVE не получит дальнейшего развития
IRP_MN_SURPRISE_REMOVAL	Уведомляет, что устройство было удалено без предварительного предупреждения

Все запросы PnP инициируются PnP Менеджером, и он всегда направляет эти запросы драйверу, находящемуся в стеке устройств на вершине стека.

Независимо от того, какие коды IRP_MN_Xxx в составе IRP запросов могут быть обработаны драйвером, те из них, которые не обрабатываются, должны быть переданы ниже по стеку устройств нижележащим драйверам, которые могут реализовывать свои собственные обработчики.

Трансляция PnP запросов вниз по стеку устройств необходима по многим причинам. Некоторые драйверы в стеке могут вносить свою лепту в обработку запроса, но не один драйвер не должен подразумевать, что запрос может быть полностью завершен на данном уровне. Например, уведомление об остановке устройства является критичным для всех слоев драйверных объектов.

Чтобы передать PnP запрос вниз, драйвер помечает IRP пакет как "завершенный" установкой соответствующих значений в полях IoStatus.Status и IoStatus.Information, а затем производит вызовы **IoCopyCurrentStackLocationToNext** и IoCallDriver. Нижележащий драйвер известен еще при выполнении AddDevice (из вызова **IoAttachDeviceToDeviceStack**), а указатель на него рекомендуется сохранять в структуре расширении объекта устройства. Пример кода, выполняющего эти действия, приводится ниже.

```
...
IoCopyCurrentIrpStackLocationToNext( pIrp );
PDEVICE_EXTENSION pThisDeviceExtension =
   (PDEVICE_EXTENSION) pThisDeviceObject->DeviceExtension;
IoCallDriver( pThisDeviceExtension ->pUnderlyingDevice, pIrp );
...
```

В случае, если у драйвера нет необходимости ожидать окончания обработки переданного вниз запроса драйверами нижних уровней, то может быть использован более эффективный механизм для пропуска участка текущего IRP стека. Функция **IoSkipCurrentIrpStackLocation** просто удаляет участок текущего стека IRP пакета из участия в обработке. Этот механизм предлагается для обращения с PnP запросами, которые не обрабатываются данным драйвером и которые должны быть просто переданы следующему нижележащему драйверу:

```
NTSTATUS
OnlyTranslateIrpDown(IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp)
{
   IoSkipCurrentIrpStackLocation( pIrp );
   PDEVICE_EXTENSION pDeviceExtension =
       (PDEVICE_EXTENSION) pDeviceObject ->DeviceExtension;
   return IoCallDriver(pThisDeviceExtension->pUnderlyingDevice, pIrp);
}
```

Бывают случаи, когда драйвер вынужден пропускать вниз PnP запросы раньше, чем он завершает собственную работу над ними. Например, при обработке запроса с кодом IRP_MN_START_DEVICE драйверу, как правило, необходимо дождаться, пока стартуют низкоуровневые драйверы перед началом работы их собственного аппаратного обеспечения. Шина и любое низкоуровневое аппаратное обеспечение инициализируется до старта отдельных устройств. Таким образом, высокоуровневые драйвера должны сначала транслировать вниз запрос и затем дождаться завершения низкоуровневой обработки перед продолжением своей работы.

Как было сказано ранее, Менеджер конфигурирования PnP ответственен за выполнение переписи устройств, обнаруженных в системе. Стартовало ли устройство при загрузке системы, или оно добавляется/удаляется позже, шинный драйвер отвечает за идентификацию и ведение списка подключенной аппаратуры. Аппаратные ресурсы, необходимые устройству, предоставляются драйверу этого устройства, когда ему отправляется сообщение (IRP пакет) с кодом IRP_MJ_PNP и с суб-кодом IRP_MN_START_DEVICE. Весьма показателен в этом отношении пример из пакета DDK, посвященный драйверам шины и устройств TOASTER.

5.2.4.3 Работа с IRP пакетами

Один из самых первых и самых важных вопросов конструирования драйвера состоит в том, как следует ли реализовать драйвер: в виде набора слоев или он должен быть монолитным.

Драйвер, реализованный по многослойной методике, имеет два преимущества. Использование слоев позволяет отделить вопросы использования высокоуровневых протоколов от вопросов, связанных с управлением собственно оборудованием. Это позволяет осуществлять поддержку аппаратуры от разных производителей без переписывания больших объемов кода. Многослойная архитектура позволяет повысить гибкость системы за счет использования при одном драйвере, реализующем протокол, сразу нескольких драйверов аппаратуры, подключаемых непосредственно во время работы. Этот прием реализован в сетевых драйверах Windows NT 5.

В случаях, когда к одному и тому же контроллеру (как это происходит в случае со SCSI адаптерами) подключаются различные типы периферийных устройств, многослойная методология позволяет отделить управление периферией от управления контроллером. Для того чтобы решить подобную проблему, необходимо написать лишь один драйвер для контроллера (порт-драйвер, *port driver*) и после этого реализовать классовые драйверы (*class driver*) для каждого типа подключаемых периферийных устройств. Две основные выгоды, получаемые здесь, состоят в том, что классовые драйвера меньше и проще, и при этом вполне вероятна ситуа-

ция, когда порт-драйвер и классовые драйвера поступают от разных поставщиков. В таком случае сборщик компьютера не обременен подбором аппаратных компонентов, имеющих общий драйвер.

Создание аппаратных шин USB и IEEE 1394 базируется на многослойном подходе к драйверам именно по перечисленным выше причинам.

Внесение драйверных слоев предоставляет простой способ добавления и удаления дополнительных возможностей устройств без установки нескольких вариантов программного обеспечения для одного и того же устройства.

Использование многослойных драйверов позволяется также скрыть от конечного пользователя некоторые аппаратные ограничения используемого устройства или ввести некоторые свойства, не поддерживаемые собственно устройством. Например, если элемент аппаратуры поддерживает работу с данными только определенного размера, то можно поставить над его драйвером другой, который будет дробить поток данных на более мелкие порции, скрывая от пользователя ограниченность возможностей этого устройства.

Разумеется, существует и оборотная сторона этой медали. Во-первых, обработка запросов ввода/вывода получает дополнительные накладные расходы, связанные с тем, что пакет IRP пропускается через код Диспетчера ввода/вывода всякий раз, когда он переходит из одного драйвера в другой. В некоторой мере эти затраты могут быть уменьшены путем введения прямого междрайверного интерфейса, который будет действовать в обход Диспетчера ввода/вывода.

Потребуются также дополнительные усилия в тестировании для того, чтобы убедиться, что отдельные компоненты получившейся драйверной конфигурации приемлемо стыкуются друг с другом. При отсутствии стандартов это может оказаться весьма болезненным этапом, особенно если драйверы поступают от разных поставщиков. Возникает также не самый простой вопрос совместимости версий разных участников получившейся иерархии.

Наконец, инсталляция многослойных драйверов также становится сложнее, поскольку каждый из них требует для себя соответствующей инсталляционной процедуры. При инсталляции необходимо установить отношения зависимости между разными драйверами в иерархии, чтобы обеспечить их запуск в правильной очередности.

Переходя к модели WDM и многослойной архитектуре, возникает необходимость уяснения практических механизмов сотрудничества драйверов в рамках этой концепции. Что должен делать драйвер, чтобы задействовать нижние драйвера и соответствовать требованиям модели, оправдывая надежды верхних драйверов.

Второе требование удовлетворить легче. Драйвер не знает и не может знать, что за клиент инициировал запрос, какова была его мотивация и полномочия. В этой ситуации драйвер просто получает IRP пакеты и должен добросовестно их обработать, отправляя их нижним слоям драйверов или выполняя всю обработку самостоятельно. Возвращая управление Диспетчеру ввода/вывода, рабочая процедура должна пометить текущий IRP пакет как завершенный (вызовом **IoCompleteRequest**) или как требующий дополнительной обработки (вызовом **IoMarkIrpPending**).

```
//Помечает пакет IRP как требующий дополнительной обработки
VOID IoMarkIrpPending(
//Указатель на текущий IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение void.

Ситуация усложняется, если речь заходит о взаимодействии с нижними драйверными слоями. Если драйвер, получая запрос от Диспетчера ввода/вывода, просто отправляет его нижним слоям (устанавливая процедуру **CompletionRoutine** перехвата пакета "на обратном пути", или не делая этого), то все сводится к тому, чтобы правильно манипулировать вызовами **IoSetCompletionRoutine**, **IoCallDriver**, **IoGetCurrentIrpStackLocation**, **IoSkipCurrentIrpStackLocation** и **IoCopyCurrentIrpStackLocationToNext**. Однако в том случае, если драйвер должен дробить поступающие запросы, накапливать, размножать (например, чтобы послать устройствам, работающим параллельно) или формировать собственные, то возникает задача создания новых пакетов IRP, поскольку только они являются средством общения между драйверами. Не составляют исключения и драйверы, работающие через прямой интерфейс (адреса вызовов), поскольку начальная инициализация интерфейса поначалу происходит через IRP запрос.

Диспетчер ввода/вывода конструирует пакеты IRP по запросу драйвера, который тот может осуществить с помощью вызовов:

- IoBuildAsynchronousFsdRequest;
- IoBuildDeviceIoControlRequest;
- IoBuildSynchonousFsdRequest.

Сконструированный пакет IRP содержит столько стековых ячеек, сколько указано в поле StackSize целевого (куда передается этот пакет IRP) объекта устройства. Указатель на целевой объект устройства передается в качестве аргумента этим перечисленным выше функциям. Таким образом,

созданные этими функциями пакеты IRP содержат достаточное количество стековых ячеек, для обеспечения вызовов всех нижележащих драйверов, но не содержит ячейки для самого промежуточного драйвера.

В случае если промежуточный драйвер использует **IoAllocateIrp** или **ExAllocatePool** для создания IRP пакета, то есть создает IRP с нуля – начиная с выделения памяти, то драйвер должен явным образом указать количество ячеек стека в пакете IRP при его создании или учесть их количество при выделении памяти. Общепринято использование поля StackSize в целевом объекте устройства для этой цели.

Как правило, у драйвера нет большой необходимости иметь собственную ячейку в стеке IRP пакета. Но в случае, если драйвер действительно испытывает необходимость в хранении собственных данных, которые потребуются в момент возвращения IRP пакета, то необходимо создавать IRP пакет, в котором будет на одну ячейку стека ввода/вывода больше – для самого драйвера. В примере, приведенном ниже показано, как это можно сделать.

```
pNewIrp = IoAllocateIrp ( pTargetDevice -> StackSize + 1, FALSE );

// Новый пакет IRP создается с указателем стека, изначально
// установленным на несуществующую позицию перед первой
// существующей ячейкой стека. Переводя указатель стека на
// одну позицию вниз, добиваемся того, что он будет указывать
// на первую ячейку, которую можно теперь использовать для
// сохранения информации, необходимой текущему (верхнему) драйверу.

IoSetNextIrpStackLocation( pNewIrp );
pUsefulArea = IoGetCurrentIrpStackLocation ( pNewIrp );
// Теперь можем использовать пространство ячейки стека вывода,
// на которую указывает pUsefulArea, по собственному усмотрению.

// Устанавливаем разнообразные необходимые значения в ячейке стека,
// соответствующей нижнему драйверу
pNextIoStackLocation = IoGetNextIrpStackLocation ( pNewIrp );
pNextIoStackLocation -> MajorFunction = IRP_MJ_XXX;
. . . . . .
// Подключаем процедуру завершения
IoSetCompletionRoutine( pNewIrp,
                        OurIoCompletionRoutine,
                        NULL, TRUE, TRUE, TRUE );

// Посылаем IRP пакет целевому драйверу:
IoCallDriver (pTargetDevice, pNewIrp );
```

В такой искусственно созданной ячейке можно хранить, например, число попыток отослать данный IRP пакет нижнему драйверу, если он отказывается обработать его немедленно. При сериализации поступившего

от клиента запроса на объемную операцию ввода/вывода, можно хранить число переданных байт данных и общий размер операции, используя один и тот же IRP пакет многократно.

5.2.4.3.1 Создание IRP пакетов вызовами IoBuild(A)SynchronousFsdRequest

Пакеты IRP можно создавать с нуля (обладая только областью памяти достаточного размера), но можно и прибегнуть к помощи рекомендованных системных вызовов **IoBuildSynchronousFsdRequest**, **IoBuildAsynchronousFsdRequest** и **IoBuildDeviceControlRequest**. Первые два вызова предназначены для конструирования IRP пакетов с кодами IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FLUSH_BUFFERS и IRP_MJ_SHUTDOWN, вполне пригодные для использования во всех драйверах, несмотря на устрашающий суффикс **Fsd**. Последний из этих вызовов, **IoBuildDeviceControlRequest**, предназначен для конструирования таких IRP пакетов, как если бы они были инициированы пользовательским API вызовом **DeviceIoControl**, то есть с кодом IRP_MJ_DEVICE_CONTROL или IRP_MJ_INTERNAL_DEVICE_CONTROL

```
//Построение IRP пакета (выделение памяти и настройка полей)
PIRP IoBuildAsynchronousFsdRequest(
//IRP_MJ_PNP или IRP_MI_READ или IRP_MJ_WRITE или IRP_MJ_FLUSH_BUFFERS или
IRP_MJ_SHUTDOWN
IN ULONG MajorFunction,
//Объект устройства, которому отдается IRP
IN PDEVICE_OBJECT pTargetDevice,
//Адрес буфера данных ввода/вывода
IN OUT PVOID pBuffer,
//Размер порции данных в байтах
IN ULONG uLenght,
//Смещение в устройстве, где начинается/продолжается операция ввода/вывода
IN PLARGE_INTEGER StartingOffset,
//Для получения завершающего статуса операций ввода/вывода
OUT PIO_STATUS_BLOCK Iosb);
```

Возвращаемое значение:

1. Не NULL — адрес нового пакета IRP
2. NULL — невозможно создать новый IRP

```
//Построение IRP пакета (выделение памяти и настройка полей)
PIRP IoBuildSynchronousFsdRequest(
//IRP_MJ_PNP или IRP_MI_READ или IRP_MJ_WRITE или IRP_MJ_FLUSH_BUFFERS или
IRP_MJ_SHUTDOWN
IN ULONG MajorFunction,
```

```
//Объект устройства, которому отдается IRP
IN PDEVICE_OBJECT pTargetDevice,
//Адрес буфера данных ввода/вывода
IN OUT PVOID pBuffer,
//Размер порции данных в байтах
IN ULONG uLenght,
//Смещение в устройстве, где начинается/продолжается операция ввода/вывода
IN PLARGE_INTEGER StartingOffset,
//Объект события, используемый для сигнализации об окончании ввода/вывода
//(должен быть инициализирован к моменту вызова). Объект переходит в сиг-
нальное состояние,
//когда нижний драйвер завершил обработку данного IRP пакета.
IN PREVENT pEvent,
//Для получения завершающего статуса операций ввода/вывода
OUT PIO_STATUS_BLOCK Iosb);
```

Возвращаемое значение:

1. Не NULL — адрес нового пакета IRP
2. NULL — невозможно создать новый IRP

Число ячеек, создаваемых в стеке ввода/вывода, размещающемся в пакете IRP, равно значению, указанному в поле pTargetDevice->StackSize. В данном случае нет простого способа создать дополнительную ячейку в стеке пакета IRP собственно для самого вызывающего драйвера.

Значения аргументов Buffer, Length и StartingOffset требуются для операций чтения и записи. Для операций flush и shutdown они должны быть установлены равными 0.

Нужные значения в области Parameters ячейки стека, соответствующей нижнему драйверу, устанавливаются автоматически, то есть нет необходимости передвигать указатель стека. Для запросов чтения или записи эти функции еще выделяют системное буферное пространство или выполняют построение MDL — в зависимости от того, выполняет ли вызываемое устройство (по указателю pTargetDevice) буферизованный или прямой ввод/вывод. При буферизованных операциях вывода производится также копирование содержимого буфера инициатора вызова в системный буфер, а в конце операции буферизованного ввода данные автоматически копируются из системного буфера в буферное пространство инициатора вызова.

Здесь общие черты этих двух функций заканчиваются. Начинаются различия.

Как следует из названия функции **IoBuildSynchronousFsdRequest**, она работает синхронно. Другими словами, поток, который выполняет вызов **IoCallDriver**, прекращает свою работу до тех пор, пока не завершится

операция ввода/вывода в нижних драйверных слоях. Для более удобной реализации такой блокировки, в создаваемый пакет IRP в виде аргумента передается адрес инициализированного объекта события (event object). Затем, после передачи созданного пакета драйверу нижнего уровня (вызовом **IoCallDriver**) следует использовать функцию **KeWaitForSingleObject** – для организации ожидания перехода этого объекта синхронизации в сигнальное состояние. Когда драйвер нижнего уровня завершит обработку данного пакета IRP, Диспетчер ввода/вывода переведет данный объект события в сигнальное состояние, что и "разбудит" данный драйвер в нужный момент. Аргумент Iosb позволяет получить информацию о том, как завершилась обработка. Заметим, что, поскольку текущий драйвер узнает о завершении обработки нового IRP пакета от функции **KeWaitForSingleObject**, то он **не должен** устанавливать свою процедуру завершения перед тем, как обратиться к нижнему драйверу вызовом **IoCallDriver**. Если же процедура завершения все-таки установлена, она всегда должна возвращать STATUS_SUCCESS.

Пакеты, созданные функцией **IoBuildSynchronousFsdRequest**, должны освобождаться только косвенно – в результате вызова **IoCompleteRequest** после получения сигнала от объекта события, а Диспетчер ввода/вывода уже сам очистит и освободит память, занятую IRP пакетом. Это включает освобождение системных буферных областей или MDL, выделенных для использования в обработке этого IRP. Использовать **IoFreeIrp** нельзя, так как такой IRP пакет участвует в очереди, организованной для пакетов, ассоциированных с данным программным потоком. Применение к нему вызова **IoFreeIrp** ранее, чем он будет удален из данной очереди, приведет к краху системы. Кроме того, во избежание неприятностей, следует следить за тем, чтобы объект события существовал к моменту, когда Диспетчер ввода/вывода соберется перевести его в сигнальное состояние.

Соответственно, фрагмент кода, который создает синхронный IRP и адресует его объекту устройства pTargetDeviceObject в нижнем драйвере, мог бы выглядеть следующим образом:

```
PIRP pIrp;
KEVENT Event;
IO_STATUS_BLOCK iosb;
KeInitializeEvent(&Event, NotificationEvent, FALSE);
pIrp = IoBuildSynchronousFsdRequest(IRP_NJ_Xxx,
                                    pTargetDeviceObject, . . .
                                    &Event,
                                    &iosb);
status = IoCallDriver(pTargetDeviceObject, pIrp);
if( status == STATUS_PENDING )
{ // Ожидаем окончания обработки в нижних слоях
```

```
   KeWaitForSingleObject(&Event, Executive, KErnelMode, FALSE,NULL);
   status = iosb.Status;
}
. . .
```

В отличие от пакетов IRP, производимых по запросу синхронной версии, функция **IoBuildAsynchronousFsdRequest** конструирует пакеты, которые не освобождаются автоматически по окончании работы над ним в нижнем драйвере. Вместо этого, драйвер, создающий "асинхронный" пакет IRP **должен обязательно** подключить свою процедуру завершения, которая и должна выполнять вызов **IoFreeIrp**. Процедура завершения и должна выполнить очистку IRP с освобождением выделенных ему системных буферных областей или MDL, а затем и освобождения памяти, занятой под структуру самого IRP пакета. В данном случае, процедура завершения должна возвратить значение STATUS_MORE_PROCESSING_REQUIRED. Соответствующий пример кода может выглядеть следующим образом:

```
. . . . .
PIRP pIrp;
IO_STATUS_BLOCK iosb;
pIrp = IoBuildAsynchronousFsdRequest(IRP_NJ_Xxx,
                                     pTargetDeviceObject, . . .
                                     &iosb);
IoSetCompletionRoutine( pIrp,
   (PIO_COMPLETION_ROUTINE) MyCompletionRoutine,
   pThisDevExtension,
   TRUE,TRUE,TRUE);
//Чтобы целевое устройство не "растворилось" за время обработки IRP:
ObReferenceObject(pTargetDeviceObject);
status = IoCallDriver(pTargetDeviceObject, pIrp);
ObDereferenceObject(pTargetDeviceObject);
. . . . .
// Процедура завершения, зарегистрированная ранее
NTSTATUS MyCompletionRoutine( PDEVICE_OBJECT pThisDevice,
                              PIRP pIrp,
                              VOID pContext )
{
   // Действия по очистке IRP
   . . . .
   IoFreeIrp( pIrp );
   return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Драйверы, которые реализуют блокирующийся механизм работы (как это получается при синхронизации по объекту события), могут привести к деградации системы. Такое может случиться, если они будут выполнять вызовы IoCallDriver с повышенных уровней IRQL. В этом случае они

могут остановиться на неопределенно долгое время, ожидая отклика с нижних уровней. Это противоречит общей философии построения Windows NT 5. Видимо, поэтому разработчики Windows искусственно затруднили построение синхронных IRP пакетов на повышенных уровнях IRQL тем, что вызов IoBuildSynchronousFsdRequest можно сделать только с уровня IRQL, равного PASSIVE_LEVEL.

Кроме того, объект события, используемый для организации ожидания, когда же закончится обработка пакета IRP на нижних уровнях, должен использоваться с максимальной осторожностью, поскольку при использовании такого драйвера в многопоточной манере могут возникнуть сложные ситуации. Допустим, два программных потока одного и того же пользовательского процесса делают запрос на запись с использованием одного и того же дескриптора (иными словами, делают запрос к одному и тому же драйверу). Тогда рабочая процедура WriteRequestHandler выполняется в контексте первого потока и останавливается в том месте, где она желает дождаться сигнала от объекта события. Затем, та же самая процедура WriteRequestHandler, выполняемая в контексте другого потока, использует повторно тот же самый объект события для обработки другого запроса. Когда запускаются оба потока, то ни один из них не может быть уверен, чей же конкретно пакет IRP обработан, поскольку при окончании обработки любого из IRP с одинаковым успехом возникает сигнал от объекта события. Решение может состоять в том, чтобы подстраховать объект события при помощи быстрого мьютекса или даже создавать новые объекты события для каждого вновь конструируемого IRP пакета.

5.2.4.3.2 Создание IRP пакетов вызовом IoBuildDeviceIoControlRequest

Последняя из упомянутых функций, предназначенных для создания IRP пакетов, IoBuildDeviceIoControlRequest также призвана облегчить этот процесс. Этот весьма полезный вызов предназначен для построения пакетов IRP, обслуживающих ввод/вывод устройств с большими вариациями в поведении и с использованием пользовательских IOCTL кодов запросов.

```
//Параметры Формирует IRP пакет (с выделением памяти),
//описывающий обращение с IOCTL запросом
PIRP IoBuildDeviceIoControlRequest(
//Код IOCTL, принимаемый (допускаемый) к обработке целевым устройством
IN ULONG IoControlCode,
//Объект устройства, которому предназначен формируемый пакет IRP
IN PDEVICE_OBJECT pTargetDevice,
//Адрес буфера ввода/вывода, передаваемого драйверу нижнего уровня
IN PVOID pInputBuffer,
//Длина буфера pInputBuffer в байтах
```

```
IN ULONG inputLenght,
//Адрес буфера ввода/вывода для данных,
//возвращаемых драйвером нижнего уровня
OUT PVOID pOutputBuffer,
//Длина буфера pOutputBuffer в байтах
IN ULONG outputLenght,
//TRUE – буден сформирован IRP пакет
//       с кодом IRP_MJ_INTERNAL_DEVICE_CONTROL
//FALSE – с кодом IRP_MJ_DEVICE_CONTROL
IN BOOLEAN InternalDeviceIoControl,
//Объект события (event object), используемый
//для сообщения об окончании ввода/вывода
IN PREVENT pEvent,
//Для получения завершающего статуса операций ввода/вывода
OUT PIO_STATUS_BLOCK pIosb);
```

Возвращаемое значение: адрес нового пакета IRP либо NULL — невозможно создать новый IRP.

Следует также отметить, что этот вызов может конструировать IRP как с синхронным способом обработки, так и асинхронным. Для получения "синхронного" IRP в функцию необходимо просто передать адрес инициализированного объекта события. После того как IRP пакет будет передан нижнему драйверу вызовом **IoCallDriver**, следует использовать **KeWaitForSingleObject** для организации ожидания сигнала от этого объекта события. Когда драйвер нижнего уровня завершит обработку IRP, Диспетчер ввода/вывода переведет объект события в "сигнальное" состояние, и в результате будет разбужен драйвер, который "организовал" весь этот процесс. Блок данных по указателю pIosb сообщает об окончательном состоянии пакета IRP. Так же, как и в случае с **IoBuildSynchronousFsdRequest**, следует аккуратнее работать в многопоточном режиме.

Диспетчер ввода/вывода автоматически выполняет очистку и освобождение IRP пакетов, созданных по вызову **IoBuildDeviceIoControlRequest** по завершении их обработки, включая подключенные к этому пакету системные буферные области или MDL. Для запуска такой очистки драйвер должен просто сделать вызов **IoCompleteRequest**.

Обычно, нет необходимости подключать процедуру завершения к пакетам IRP такого типа, если только у драйвера нет необходимости выполнить какие-нибудь специфические действия пост-обработки. Но уж если такая процедура подключена, то она должна возвращать значение STATUS_SUCCESS, чтобы позволить Диспетчеру ввода/вывода выполнить очистку этого пакета по окончании процедуры завершения.

Метод буферизации, который указан в IOCTL коде, влияет на формирование IRP пакета. В том случае, если IOCTL код описан как METHOD_BUFFERED, внутри вызова IoBuildDeviceIoControlRequest выполняется выделение области нестраничной памяти, куда производится копирование содержимого буфера по адресу pInputBuffer. Когда обработка IRP завершается, содержимое буфера в нестраничном пуле автоматически копируется в область памяти по адресу pOutputBuffer.

В случае, если IOCTL код содержит флаги METHOD_OUT_DIRECY или METHOD_IN_DIRECT, то **IoBuildDeviceIoControlRequest всегда** выполняет построение MDL списка для буфера pOutputBuffer и всегда использует буфер в нестраничной памяти для буфера pInputBuffer, **независимо** от того, указан ли METHOD_IN_DIRECT или METHOD_OUT_DIRECT. В общем-то, формирование IRP пакета в обоих случаях происходит совершенно аналогично тому, как если бы в Win32 обрабатывался вызов **DeviceIoControl**, поступивший из приложения пользовательского режима.

5.2.4.3.3 Создание IRP пакетов "с нуля"

В отдельных редких случаях, когда нужно формировать запрос, отличающийся от операций чтения, записи, очистки буферов, операции shutdown или IOCTL операций, единственным вариантом остается выделение памяти под пакет IRP и заполнение его нужными данными "вручную".

Для формирования пакетов IRP можно использовать функцию **IoAllocateIrp**, которая выполняет выделение памяти под пакет IRP в зонном буфере Диспетчера ввода/вывода, после чего выполняет некоторые действия по инициализации полей в выделенной области.

Попробуем отказаться и от этой услуги Диспетчера ввода/вывода и создать пакет IRP "совершенно с нуля" на примере IRP пакета для буферизованного ввода/вывода.

В данном случае память под IRP пакет выделяется в нестраничном пуле при помощи вызова **ExAllocatePool**, а затем производится инициализация необходимых полей внутри созданной области. Общая инициализация выделенной области по типу "IRP пакет" должна быть выполнена при помощи вызова **IoInitializeIrp**. Установка полей в той ячейке стека IRP пакета, которую будет разбирать драйвер-получатель (владеющий устройством pTargetDevice), и буферных областей для передачи данных возлагается на текущий драйвер.

Предполагается, что текущий драйвер получил IRP пакет pOriginalIrp и должен сформировать IRP пакет для запроса на чтение (хотя именно его проще было бы сформировать описанными ранее вызовами).

```
. . . . .
#define BUFFER_SIZE (1024)
CCHAR nOfRequiredStackLocs = pTargetDevice->StackSize;
USHORT irpSize = IoSizeOfIrp(nOfRequiredStackLocs);
PIO_STACK_LOCATION pTagDevIrpStackLocation;

PIRP pCreatedIrp = (PIRP) ExAllocatePool( NonPagedPool, irpSize );
IoInitializeIrp( pCreatedIrp, irpSize, nOfRequiredStackLocs);

// Получаем указатель на ячейку стека IRP, которая после вызова
// IoCallDriver будет ассоциирована с нижним драйвером:
pTagDevIrpStackLocation = IoGetNextIrpStackLocation( pCreatedIrp );

// Подразумевая операцию чтения, устанавливаем поля ячейки:
pTagDevIrpStackLocation->MajorFunction = IRP_MJ_READ;
pTagDevIrpStackLocation->Parameters.Read.Length = BUFFER_SIZE;
pTagDevIrpStackLocation->Parameters.Read.ByteOffset.QuadPart = 0i64;

// В запросе IRP_MJ_READ список MDL не может использоваться.
// Передаем собственный буфер в качестве системного,
// требующегося при данном типе запросов:
PVOID newBuffer = ExAllocatePool ( NonPagedPool, BUFFER_SIZE );
pCreatedIrp -> AssociatedIrp.SystemBuffer = newBuffer;

// Если вызываемое устройство имеет свойство (флаг) DO_DIRECT_IO:
if( pTargetDevice->Flags & DO_DIRECT_IO )
{
   // Описание IoAllocateMdl см. в таблице 7.19. Поскольку третий
   // параметр равен FALSE, указатель на созданный MDL список будет
   // сразу занесен в поле pCreatedIrp-> MdlAddress
   PMDL pNewMdl = IoAllocateMdl ( newBuffer,
                                  BUFFER_SIZE,
                                  FALSE, FALSE,
                                  pCreatedIrp);

   // для буфера в нестраничной памяти:
   MmBuildMdlForNonPagedPool( pNewMdl);
}

// Копируем информацию о потоке инициатора вызова:
pCreatedIrp -> Tail.Overlay.Thread =
pOriginalIrp -> Tail.Overlay.Thread;

// Устанавливаем процедуру завершения обработки сформированного IRP
IoSetCompletionRoutine ( pCreatedIrp,
                         MyIoCompletionRoutine,
                         NULL, TRUE, TRUE, TRUE );

// Передаем созданный пакет драйверу нижнего уровня
IoCallDriver ( pTargetDevice, pCreatedIrp );
```

Неочевидность приведенных выше манипуляций с ячейкой стека IRP пакета говорит о том, что необходимо в совершенстве владеть тонкостями формирования пакетов для тех типов запросов, которые вам захочется создавать самостоятельно.

В процедуре завершения следует переместить полученные "снизу" данные соответствующему получателю наверху. Разумеется, в процедуре завершения необходимо выполнить и действия по освобождению ресурсов, присвоенных IRP пакету, то есть выполнить освобождение памяти, занятой для системного буфера и собственно IRP пакета.

```
NTSTATUS MyIoCompletionRoutine(IN PDEVICE_OBJECT pThisDeviceObject,
                               IN PIRP  pIrp,
                               IN PVOID pContext )
{
. . .
// Очистка структуры MDL списка:
IoFreeMdl( pIrp->MdlAddress );

// Освобождение специального буфера:
IoFreePool ( pIrp->AssociatedIrp.SystemBuffer );

// Освобождение собственно IRP:
IoFreeIrp ( pIrp );

return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Возможны ситуации, когда драйвер ведет собственную политику относительно выделения памяти под IRP пакеты, например, из ассоциативного списка, им же созданного. Такие IRP все равно должны быть инициализированы вызовом **IoInitializeIrp**, однако, применять к ним вызов **IoFreeIrp** нельзя, поскольку тот, скорее всего, нарушит учет памяти в драйвере.

При всей сложности самостоятельного создания IRP пакетов с нуля, в этом есть одно важное преимущество – драйвер контролирует количество создаваемых ячеек стека IRP пакета. В том числе – дополнительных, которые могут оказаться в некоторых случаях незаменимыми для хранения специфичной (для данного IRP пакета и для данного драйвера) информации в течение времени жизни этого IRP пакета.

Ниже приводятся описания прототипов использованных функций.

```
//Формирует IRP пакет с выделением памяти
//(не требует последующего вызова IoInitializeIRP)
PIRP IoAllocateIrp(
//Количество ячеек стека во вновь создаваемом IRP пакете
IN CCHAR StackSize,
```

```
//FALSE
IN BOOLEAN ChargeQuota);
```

Возвращаемое значение Адрес нового пакета IRP либо NULL – невозможно создать новый IRP.

```
//Формирует IRP пакет в ранее выделенной области памяти
//(не должна использоваться для пакетов, созданных вызовом IoAllocateIrp)
VOID IoInitializeIrp(
//Указатель на область, используемую под IRP
IN PIRP pIrp,
//Заранее вычисленный общий размер IRP пакета
//(можно использовать вызов IoSizeOfIrp)
IN USHORT PacketSize,
//Количество ячеек стека во вновь создаваемом IRP пакете
IN CCHAR StackSize);
```

Возвращаемое значение void.

```
//Очищает и освобождает IRP пакеты, созданные вызовами
//IoAllocateIrp или IoBuildAsynchronousFsdRequest
VOID IoFreeIrp(
//Указатель на освобождаемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение void

Как было сказано ранее, пакеты, созданные **IoBuildSynchronousFsdRequest** или **IoBuildDeviceIoControlRequest**, освобождаются самим Диспетчером ввода/вывода, когда драйвер завершает обработку такого пакета вызовом **IoCompleteRequest**. Освобождения пакетов, сделанных нестандартными способами (например, с помощью ExAllocatePool) выполняет сам драйвер.

```
//Определяет размер IRP пакета, как если бы он имел StackSize ячеек стека
USHORT IoSizeOfIrp(
//Предполагаемое число ячеек стека IRP пакета
IN CCHAR StackSize);
```

Возвращаемое значение Размер в байтах.

5.2.4.3.4 Работа с IRP пакетами-репликантами

Если промежуточный драйвер занимается "выпуском" собственных IRP пакетов, направляя при этом нижним драйверным слоям (одному или нескольким разным драйверам) по нескольку экземпляров сразу, то он ра-

но или поздно попадет в следующую непростую ситуацию. До момента окончания обработки всех "вторичных" пакетов драйвер не может завершить обработку исходного (изначально поступившего) IRP пакета. Возможны два способа работы с размноженными вторичными пакетами.

В первом, "синхронном", случае рабочая процедура должна дождаться, пока все созданные драйвером IRP пакеты не будут обработаны в нижних уровнях. В общем случае, рабочая процедура выполняет следующее:

1) Выполняет вызовы **IoBuildSynchronousFsdRequest** или вызовы **IoBuildDeviceIoControlRequest** для того, чтобы создать необходимое количество IRP пакетов "синхронного" типа.
2) Выполняет вызовы **IoCallDriver** для передачи всех созданных драйвером пакетов IRP другим драйверам.
3) Выполняет вызовы **KeWaitForMultipleObjects** и ожидает завершения обработки всех переданных IRP пакетов.
4) Выполняет действия по переносу информации из полученных пакетов и их последующую очистку и освобождение.
5) Наконец, выполняет вызов **IoCompleteRequest** относительно исходного IRP пакета для того, чтобы возвратить его инициатору вызова.

Поскольку исходный запрос удерживается внутри рабочей процедуры (поскольку она не возвращает управление), то нет и необходимости помечать исходный IRP пакет как ожидающий обработки.

Второй, "асинхронный", случай несколько сложнее, поскольку непонятно, где именно драйвер может остановиться и дожидаться завершения обработки всех пакетов. В этом случае драйверу рекомендуется подключить процедуру завершения к каждому созданному им IRP пакету, и процедура завершения (скорее всего — единственная для всех пакетов) должна сама определить, наступило ли время считать, что обработка исходного IRP запроса действительно завершена. План действий примерно таков:

1) Пометить пакет IRP, поступивший в рабочую процедуру от Диспетчера ввода/вывода как ожидающий обработки при помощи **и**.
2) Создать дополнительные пакеты IRP с использованием одного из описанных выше методов.
3) Подключить процедуру завершения (возможно — одну и ту же) к каждому из вновь созданных IRP пакетов вызовом **IoSetCompletionRoutine**. При выполнении этого вызова следует передать указатель на исходный IRP пакет в аргументе pContext.
4) Запомнить число созданных пакетов IRP в неиспользуемом поле исходного IRP пакета. Поле Parameters.Key текущей ячейки стека IRP пакета вполне годится.

5) Передать пакеты всем нужным драйверам вызовом **IoCallDriver**.
6) Возвратить значение STATUS_PENDING, поскольку обработка исходного запроса (пакета IRP) не завершена.

По окончании обработки каждого IRP пакета драйвером нижнего уровня во втором, "асинхронном", варианте вызывается процедура завершения рассматриваемого ("нашего") драйвера, которая выполняет следующие операции:

1) Выполняет необходимый перенос информации, очистку и удаление созданного драйвером IRP пакета, вернувшегося от нижнего драйвера.
2) Уменьшает на единицу сохраненное ранее число незавершенных пакетов IRP. Это действие рекомендуется выполнять, приняв хотя бы минимальные меры по безопасному доступу к этому значению. Вполне подходит для этой цели вызов **InterlockedDecrement**.
3) В случае, если незавершенных пакетов не осталось, выполняет вызов **IoCompleteRequest**, что сигнализирует о полном завершении обработки исходного IRP запроса.
4) Возвращает управление Диспетчеру ввода/вывода с кодом завершения STATUS_MORE_PROCESSING_REQUIRED — для того, чтобы не допустить вызов процедур завершения вышестоящих драйверов для работы над пришедшим "снизу" IRP пакетом, созданным данным драйвером. Кстати заметить, к этому моменту рассматриваемый IRP пакет уже уничтожен.

5.2.4.3.5 Удаление IRP пакетов

Как бывает и в реальной жизни, кто-то, инициировавший IRP запрос, может передумать и инициализировать снятие запроса "с повестки". Пользовательское приложение может запросить уничтожение пакета после длительного ожидания. Приложение может вовсе прекратить работу, бросив все на попечение операционной системы. Наконец, приложение может попытаться завершить свою асинхронную операцию Win32 API вызовом **CancelIo**.

В режиме ядра для удаления запроса выполняется вызов **IoCancelIrp**. Операционная система также вызывает **IoCancelIrp** для всех IRP пакетов, относящихся к потоку, выполнение которого прекращается.

```
//Помечает пакет IRP как требующий удаления
//и вызывает процедуры удаления, если таковые определены
BOOLEAN IoCancelIrp(
```

```
//Указатель на удаляемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение: TRUE – если пакет удален или FALSE – в случае неудачи.

Предположим, некий код режима ядра направил пакет (в данном случае – синхронный) другому драйверу. Как он может выполнить удаление отправленного пакета, например, в результате превышения времени ожидания? Пример ниже иллюстрирует этот случай.

```
// формирует синхронный пакет:
PIRP pIrp= IoBuildSynchronousFsdRequest(. . ., &event, &iosb);
// Подключаем процедуру завершения:
IoSetCompletionRoutine( pIrp, MyCompletionRoutine, (VOID*)&event,
                        TRUE, TRUE, TRUE );
NTSTATUS status = IoCallDriver(. . .);
if( status == STATUS_PENDING )
{  // Некоторое время ожидаем естественного завершения
   LARGE_INTEGER waitDelay;
   waitDelay.QuadPart = - 10000; // относительное время
   if( KeWaitForSingleObject( &event,
       KernelMode, FALSE, &waitDelay) == STATUS_TIMEOUT )
   {
       IoCancelIrp(pIrp);
       KeWaitForSingleObject( &event, KernelMode, FALSE, NULL);
   }
}
// Синхронные IRP пакеты - их удаляет Диспетчер ввода/вывода:
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
. . .

// Процедура завершения
NTSTATUS MyCompletionRoutine( PDEVICE_OBJECT pThisDevice,
                              PIRP pIrp,
                              PVOID pContext )
{
   if (pIrp->PendingReturned)
       KeSetEvent((PKEVENT) pContext, IO_NO_INCREMENT, FALSE);
   return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Процедура **IoCancelIrp** устанавливает флаг (cancel bit) в IRP пакете и выполняет вызов процедуры **CancelRoutine**, если таковая имеется в IRP пакете.

В отличие от процедур завершения, которые могут быть указаны для каждого драйвера (в соответствующих ячейках стека IRP), место для процедуры **CancelRoutine** в IRP пакете единственное, что указывает на ее необычный статус — она вызывается один раз для удаляемого пакета. Соответственно, эта процедура и должна принадлежать тому драйверу, который наиболее компетентно может распорядиться судьбой IRP пакета. Правда, этот драйвер может регистрировать разные свои функции для удаления разнотипных IRP пакетов (чтения, записи и т.п.).

Драйвер должен участвовать в схеме удаления IRP пакетов, если он реализует собственную очередь пакетов или участвует в использовании системной очереди (то есть зарегистрировал процедуру StartIo). При этом подразумевается, что процедуре удаления подвергаются прежде всего пакеты в состоянии ожидания, то есть не выполненные сразу, а помещенные по этой причине в очередь. Пакеты IRP, которые переданы на обработку нижним драйверам и "застряли" там — это худшее, что можно придумать в момент удаления.

```
//Выполняет действия, сопутствующие удалению пакета IRP
VOID CancelRoutine(
//Указатель на объект устройства, которое (точнее – драйвер)
//и зарегистрировало ранее эту функцию в IRP пакете вызовом
IoSetCancelRoutine
IN PDEVICE_OBJECT pDevObj,
//Указатель на удаляемый IRP пакет
IN PIRP pIrp);
```

Возвращаемое значение void.

```
//Устанавливает (переустанавливает) определяемую драйвером
//функцию CancelRoutine для данного IRP пакета
PDRIVER_CANCEL IoSetCancelRoutine(
//Указатель на IRP пакет, которому будет соответствовать
//устанавливаемая функция CancelRoutine
IN PIRP pIrp,
//Указатель на функцию, которая соответствует прототипу, или NULL
//(если следует отменить функцию CancelRoutine для данного пакета IRP)
IN PDRIVER_CANCEL CancelRoutine);
```

Возвращаемое значение: указатель на ранее установленную для данного IRP пакета функцию CancelRoutine. Соответственно, если таковой не было, то возвращается NULL. Значение NULL возвращается также, если пакет находится в обработке и не может быть удален.

Для ограничения доступа к удаляемому пакету, код **IoCancelIrp**, прежде всего, запрашивает объект спин-блокировки вызовом **IoAcquireCancelSpinLock** (в переменной pIrp->CancelIrql сохраняется зна-

чение текущего уровня IRQL для использования при последующем вызове **IoReleaseCancelSpinLock**). В случае если за IRP пакетом закреплена процедура **CancelRoutine**, то она вызывается (теперь на нее возложена задача освобождения спин-блокировки). Если же такой процедуры нет, то вызов IoCancelIrp завершает работу, освобождая спин-блокировку.

Процедура CancelRoutine, получив управление, может оказаться в одной из описанных ниже ситуаций.

Во-первых, рассматриваемый IRP пакет в настоящий момент обрабатывается. Если такой пакет все еще находится в одной из рабочих процедур драйвера, то он имеет право на несколько секунд жизни, после чего драйвер должен принять решение о его дальнейшей судьбе, скорее всего, завершить с ошибкой, например:

```
Irp->IoStatus.Status = STATUS_IO_TIMEOUT;
Irp->IoStatus.Information = 0;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

В том случае, если пакет "застрял" в нижних слоях драйверов, то это вина нижних драйверов, которые не смогли организовать обработку данной нештатной ситуации. Лучшим приемом будет — дождаться естественного развития событий, когда пакет вернется, вероятнее всего, с каким-нибудь кодом ошибки.

Выяснить, обрабатывается ли рассматриваемый пакет именно сейчас, можно при помощи следующего кода, поскольку, если задействован механизм System Queuing и какой-либо пропущенный через него IRP пакет в настоящий момент обрабатывается, то именно адрес этого IRP пакета "лежит" в поле pDeviceObject->CurrentIrp (иначе там будет NULL):

```
VOID MyCancelRoutine ( IN PDEVICE_OBJECT pDeviceObject,
                       IN PIRP pIrp)
{
   if( pIrp == pDeviceObject->CurrentIrp )
   {
   . . .
```

Конкретная реализация действий по удалению текущего пакета (если она возможна) остается задачей разработчика драйвера.

Существенно проще становится ситуация, когда пакет, предназначенный для уничтожения, только что поступил в процедуру StartIo либо еще находится в очереди отложенных (pending) пакетов.

```
VOID StartIo ( IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIRp)
{
   KIRQL CancelIrql;
```

```
    IoAcquireCancelSpinLock(&CancelIrql);
    If(pIrp->Cancel)
    {
        IoReleaseCancelSpinLock(CancelIrql);
        return;
    }
    // Удаляем процедуру обработки удаления, делая пакет
    // "not cancelable" - неуничтожаемым
    IoSetCancelRoutine(pIrp, NULL);
    IoReleaseCancelSpinLock(CancelIrql);
    . . .
}
```

В случае, если удаляемый IRP пакет пока находится в системной очереди (которая называется еще "управляемая StartIo"), **а перед его размещением** там (то есть вместе с вызовом **IoMarkIrpPending**) была зарегистрирована процедура MyCancelRoutine для этого IRP пакета, то действия по удалению такого пакета (не текущего — не находящегося в обработке) могут выглядеть следующим образом:

```
VOID MyCancelRoutine( IN PDEVICE_OBJECT pDeviceObject, IN PIRP pIrp)
{
    if( pIrp == pDeviceObject->CurrentIrp )
    {   // Текущий IRP
        IoReleaseCancelSpinLock(pIrp->CancelIrql);
        // Вряд ли можно сделать что-то еще...
    }
    else
    {   // Удаляем из системной очереди:
        KeRemoveEntryDeviceQueue( &pDeviceObject->DeviceQueue,
            &pIrp->Tail.Overlay.DeviceQueueEntry);
        // Только теперь можно освободить спин-блокировку:
        IoReleaseCancelSpinLock(pIrp->CancelIrql);

        pIrp->IoStatus.Status=STATUS_CANCELLED;
        pIrp->IoStatus.Information = 0;
        IoCompleteRequest( pIrp, IO_NO_INCREMENT );
    }
    return;
}
```

Приведенный ниже пример выполняет удаление пакета из очереди, поддерживаемой собственно драйвером (так называемой "Device-Managed Queue").

```
VOID MyOtherCancelRoutine ( IN PDEVICE_OBJECT pDeviceObject,
                            IN PIRP pIrp )
{
    KIRQL oldIRQL;
```

```
    PMYDEVICE_EXTENSION pDevExt =
        (PMYDEVICE_EXTENSION)pDeviceObject->DeviceExtension;
        IoSetCancelRoutine(pIrp, NULL);
    // Освобождаем спин-блокировку, установленную еще IoCancelIrp
    IoReleaseCancelSpinLock(pIrp->CancelIrp);

    // Удаляем IRP из очереди под защитой спин-блокировки
    KeAcquireSpinLock(&pDevExt->QueueLock, &oldIRQL);
    RemoveEntryList (&pIrp->Tail.Overlay.ListEntry);
    KeReleaseSpinLock(&pDevExt->QueueLock, oldIRQL);
    //
    pIrp->IoStatus.Status = STATUS_CANCELLED;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest( pIrp, IO_NO_INCREMENT );
    return;
}
```

Предполагается, что объект спин-блокировки, используемый для синхронизации доступа к очереди пакетов, pDevExt->QueueLock был заранее создан и сохранен в структуре расширения данного устройства.

В заключение, следует отметить, что после вызова **IoCompleteRequest** в конце процедуры CancelRoutine (обработки удаления IRP пакета в данном драйвере) запускаются вызовы процедур завершения вышестоящих драйверов — если такие драйвера имеются и если соответствующие процедуры были зарегистрированы для данного IRP пакета на случай его удаления.

5.3 Разработка драйверов для ОС Linux

Стандартные исходные драйверы ОС Linux можно просмотреть в каталоге исходных текстов Linux drivers:

input/keyboard/atkbd.c – клавиатура;
input/mouse/ – «мышь»;
input/mouse/synaptics.c – сенсорная панель ноутбука;
input/gameport/gameport.c, emu10k1-gp.c – игровой порт;
ieee1394/ieee1394_core. c – FireWire (IEEE1394);
drivers/pci/pci.c – шины PCI;
i2c/i2c-core.c – шины I2C;
net/ne2k-pci.c – сетевой карты NE2000;
ide/ide.c – IDE-интерфейса.

Драйверы модульных ОС типа Unix имеют определенную структуру и принципы функционирования и использования. Ознакомиться с ними можно, в документе «The Linux Kernel Module Programming Guide» в оригинале или в переводе на русский язык http://gazette.linux.ru.net/rus/articles/lkmpg.html.

5.3.1 Драйвера и строение ядра Linux

Драйверы играют особую роль в ядре Linux. Это настоящие "черные ящики", которые полностью скрывают детали, касающиеся работы обслуживаемого устройства, и предоставляют четкий программный интерфейс для работы с аппаратурой. Действия пользователя обрабатываются набором стандартизированных запросов, которые не зависят от определенного драйвера. Роль драйвера заключается в том, чтобы преобразовать эти запросы в операции, специфичные для заданного устройства, посредством которых и осуществляется взаимодействие с аппаратурой. Программный интерфейс задуман так, что он позволяет собирать драйверы отдельно от ядра и "подключать" их во время работы по мере необходимости. Такая модульная организация значительно облегчает написание драйверов в Linux.

Каждый драйвер имеет определенные отличия от других и разработчик драйвера, должен четко знать специфику устройства, для которого создается драйвер. Тем не менее, основные правила и методы разработки одинаковы для большинства случаев. Программист постоянно должны будете идти на компромисс, выбирая между временем, затрачиваемым на программирование и универсальностью драйвера.

Разделение тактики и стратегии – одна из замечательных черт дизайна Unix. Значительная часть задач может быть разбита на две части: "какие возможности следует обеспечить" (тактика) и "как эти возможности использовать" (стратегия). Если эти две подзадачи решаются в разных частях программы или даже разными программами, то такой программный пакет намного проще развивать и адаптировать под изменяющиеся потребности.

Например, управление графическим дисплеем в Unix производит X-сервер, который работает с аппаратным окружением и предоставляет обобщенный инерфейс для пользовательских программ. Оконный менеджер осуществляют стратегию, ничего не зная об имеющихся аппаратных средствах. Разные люди могут использовать один и тот же оконный менеджер на различной аппаратуре, а различные пользователи могут использовать разные оконные менеджеры на одной и той же рабочей станции. Даже такие, отличные друг от друга окружения рабочего стола, как KDE и GNOME могут прекрасно сосуществовать вместе. Другой пример – многоуровневая архитектура стека протоколов TCP/IP. Операционная система предоставляет абстракцию сокетов, которая не реализует никакой стратегии по обработке получаемых или передаваемых данных, в то время как за предоставление конкретных сетевых услуг отвечают конкретные серверы (реализующие свои стратегии).

Такое же разделение относится и к драйверам. Драйвер floppy не реализует стратегии. Его назначение – представить дискету как непрерывный

массив блоков данных. Более высокие уровни операционной системы реализуют стратегию работы с дискетой – обращаться ли к ней напрямую или посредством файловой системы, а также решают вопрос – могут ли пользователи монтировать файловую систему на дискете. Поскольку разные среды работают с аппаратурой по-разному, очень важно, чтобы драйверы были свободны от принятия стратегических решений настолько, насколько это возможно.

При написании драйвера, программист должен обращать особое внимание на фундаментальный аспект: код ядра, предоставляющий доступ к аппаратуре, не должен заниматься реализацей стратегических решений для пользователей, поскольку пользователи могут иметь совершенно различные потребности. Драйвер должен обеспечивать доступ к аппаратуре, оставляя решение проблемы о том, как ее использовать прикладным программам. Универсализм драйвера заключается в том, чтобы обеспечить доступ к возможностям аппаратуры не накладывая дополнительных ограничений. Иногда, однако, приходится выполнять реализацию отдельных стратегических решений. Например, драйвер дискретного ввода-вывода сможет получать-выдавать данные только побайтно, если не добавить дополнительный код, который будет обрабатывать отдельные биты.

На драйвер можно взглянуть и под другим углом: это промежуточный уровень между приложением и аппаратурой. Такое привелигированное положение позволяет программисту сделать выбор, как будет представлено данное устройство: различные драйверы могут предоставлять различные возможности даже в случае одного и того же устройства.

Драйверы, свободные от стратегических решений, несут в себе множество характерных особенностей. Среди них – поддержка как синхронных, так и асинхронных операций; возможность работы в многозадачной среде; максимально полное использование возможностей аппаратуры; отсутствие стратегической прослойки, что упрощает код драйвера, оставляя его простым и понятным. Драйверы такого сорта не только лучше работают, но и более просты в разработке и сопровождении.

В операционной системе Unix одновременно работает множество *процессов*, выполняющих самые разнообразные задачи. Каждый процесс запрашивает у системы различные ресурсы, будь то процессор, память, сеть или нечто иное. Принимает и обрабатывает все эти запросы – *ядро*, которое является одним большим испоняемым файлом. Хотя границы различных задач, выполняемых ядром, не всегда могут быть четко проведены, тем не менее функциональность ядра может быть разделена на несколько частей, как показано на рисунке 3.

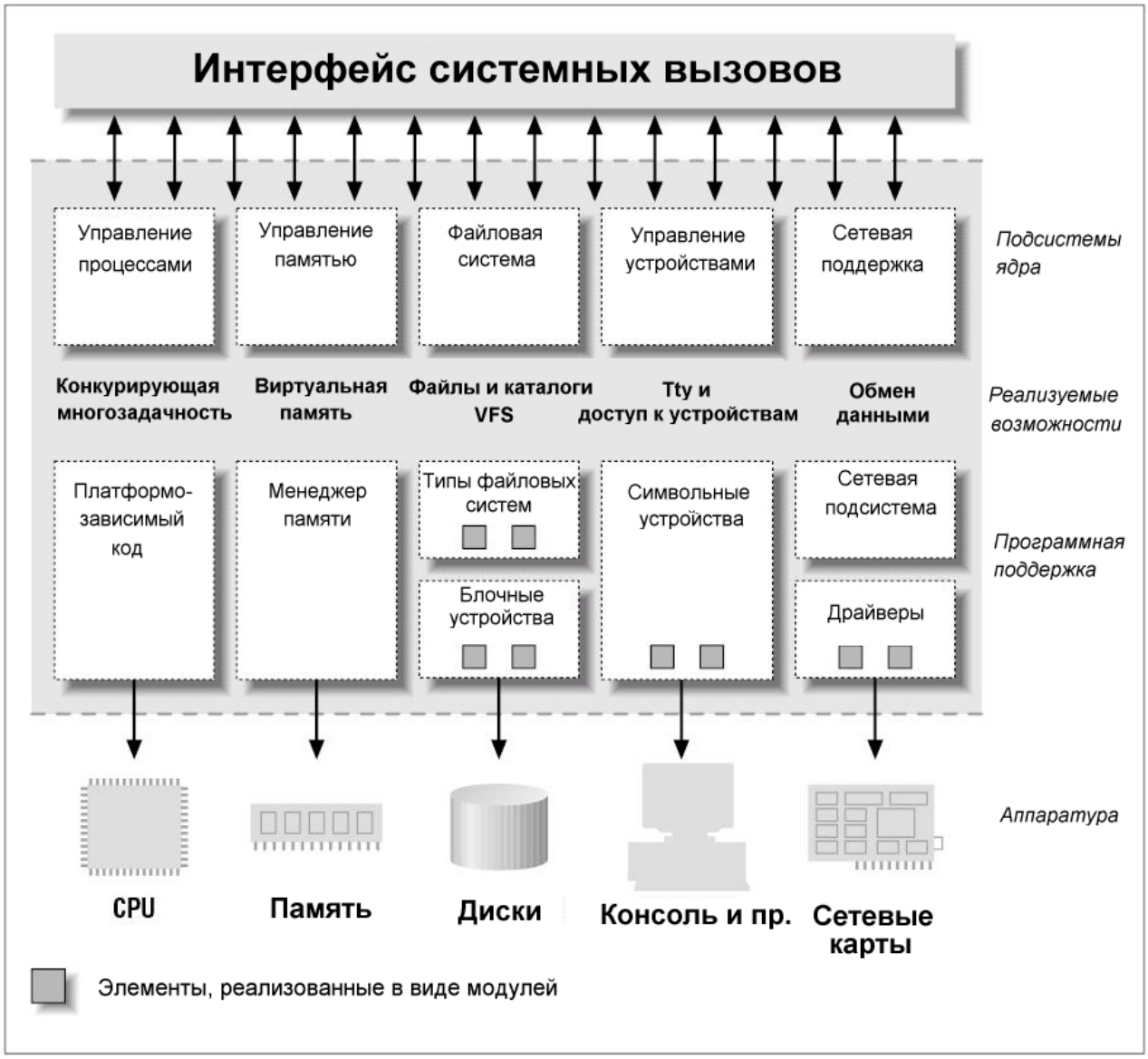

Рисунок 3 – Строение ядра

Традиционно, Unix разделяет все устройства на три основных типа. Каждый модуль, как правило, реализует функциональность одного из этих типов и отсюда может классифицироваться как символьный модуль, блочный модуль или сетевой модуль. Это деление не жесткое, программист может создать один большой модуль, в котором будет реализовано несколько различных драйверов. Но обычно хорошие программисты создают отдельные модули для каждого случая, поскольку декомпозиция – есть ключевой момент масштабируемости и расширяемости.

Ниже приводятся краткие описания этих трех классов:

1. Символьное устройство – это такое устройство, которое поставляет данные как поток байт (напоминает файлы). Драйвер символьного устройства отвечает за реализацию поддержки такого поведения. Такие драйверы реализуют по меньшей мере четыре системных вызова: open,

close, write и read. Типичными примерами таких устройств могут служить текстовая консоль (/dev/console) и последовательный порт (/dev/ttyS*). К символьным устройствам можно обращаться посредством элементов файловой системы, таких как /dev/tty1 или /dev/lp0. Единственное важное отличие обычных файлов от символьных устройств заключается в том, что вы можете перемещаться по файлам взад и вперёд, в то время как символьные устройства – это лишь каналы передачи данных, доступ к которым осуществляется в заданном порядке. Тем не менее, существуют символьные устройства, которые допускают произвольный доступ к данным в потоке, типичным примером могут служить устройства захвата изображения, где приложения могут обращаться к изображению целиком, используя mmap или lseek.

2. Блочные устройства – подобно символьным устройствам, позволяют обращаться посредством элементов файловой системы в каталоге /dev. Самым известным примером блочного устройства может служить жёсткий диск. Обмен данными с блочным устройством производится порциями байт – блоками. В большинстве Unix-систем размер одного блока равен 1 килобайту или другому числу, являющемуся степенью числа 2. Linux позволяет приложениям обращаться к блочному устройству так же как к символьному – он разрешает передачу любого числа байт в блоке. В результате, все различие между блочными и символьными устройствами сводится к внутреннему представлению данных в ядре. Драйвер блочного устройства реализует точно такой же интерфейс с ядром, что и драйвер символьного устройства, но дополнительно реализуется еще и блочно-ориентированный интерфейс, который "невидим" для пользователя или приложения, которые открывают доступ к блочному устройству посредством псевдофайловой системы /dev. Тем не менее, блочный интерфейс совершенно необходим, чтобы можно было выполнить mount файловой системы.

3. Сетевые интерфейсы. Любой сетевой обмен выполняется через сетевой интерфейс, т.е. устройство, которое способно обмениваться данными с другими узлами сети. Как правило – это некое аппаратное устройство, но возможна и исключительно программная реализация сетевого устройства, например петлевое устройство loopback (локальный сетевой интерфейс). Сетевой интерфейс отвечает за передачу и получение пакетов данных, которыми управляет сетевая подсистема ядра, ничего не зная о том, к каким соединениям эти пакеты принадлежат. Не смотря на то, что соединения по протоколам Telnet и FTP используют один и тот же сетевой интерфейс, само устройство не различает эти соединения, оно "видит" только пакеты данных. Не будучи потоковым устройством, сетевой интерфейс не может быть отображен на файловую систему, как это делается в случае с /dev/tty1. Традиционно Unix предоставляет доступ к интерфейсу,

назначая ему уникальное имя (например, eth0), но вы не найдете файл с этим именем в каталоге /dev. Принципы взаимодействия ядра с драйвером сетевого устройства, в корне отличаются от принципов, применимых к символьным или блочным устройствам. Вместо функций write и read, ядро вызывает соответствующие функции, управляющие передачей пакетов.

В Linux существуют и другие классы модулей драйверов, которые добавляют в ядро возможность взаимодействия с определенными типами устройств. Из таких нестандартных классов устройств наиболее часто встречается SCSI (Small Computer Systems Interface). Хотя внешние устройства, связанные с шиной SCSI, и отображаются в каталоге `/dev`, в виде символьных или блочных устройств, тем не менее, как и в случае с сетевыми интерфейсами, внутренняя программная организация отличается очень сильно. Сюда же можно отнести USB, FireWire и I2O.

Помимо драйверов устройств, пожалуй, самым важным классом модулей в Linux являются файловые системы. Тип файловой системы обусловливает способ организации информации на блочном устройстве. Это – не драйвер устройства, здесь нет никакого устройства, которое было бы связано со способом размещения информации. Тип файловой системы – это программный драйвер, потому что он отображает структуры данных нижнего уровня на структуры данных верхнего уровня. Файловая система определяет какой длины могут быть имена файлов, и какая информация о каждом из файлов должна храниться. Файловая система должна реализовать самый нижний уровень системных вызовов для доступа к каталогам и файлам, путем отображения их имён (и иной информации, такой как права доступа и пр.) в структуры данных, которые записываются в блоки данных. Такой интерфейс полностью независим от физической передачи данных с/на диск (или другой носитель), которая выполняется драйвером блочного устройства. Linux поддерживает модули файловых систем, которые реализуют различные файловые операции. Однако вам едва ли придется писать свой модуль файловой системы, поскольку ядро Linux включает в себя код довольно большого числа наиболее популярных файловых систем.

5.3.2 Сборка и запуск модулей

Модуль выполняется в так называемом **пространстве ядра**, в то время как прикладные задачи – в **пространстве пользователя**. Это базовая концепция в теории построения операционных систем.

Роль операционной системы, фактически состоит в предоставлении аппаратных ресурсов в распоряжение прикладных программ. Кроме того, операционная система должна быть изолирована от действий прикладных программ и предотвращать несанкционированный доступ к ресурсам компьютера.

Любой современный процессор может предложить защиту в виде нескольких уровней привелегий. Каждый из уровней играет свою роль, и некоторые операции запрещены к исполнению на более низких уровнях. Программный код может переходить с одного уровня на другой ограниченным числом способов. Unix системы спроектированы так, что они в состоянии использовать в своих интересах два таких уровня. Все современные процессоры имеют, по меньшей мере, два уровня защиты, а в некоторых, например x86 – их больше. При наличии у процессора более чем двух уровней защиты используются только два – высший и низший. Ядро Linux исполняется на высшем уровне (он еще называется привелигированный режим), где позволяется выполнение любых действий. Приложения же исполняются на самом нижнем уровне (так называемый непривелигированный режим), где прямой доступ к аппаратуре и памяти регулируется процессором.

Обычно, о режимах исполнения, мы говорим как пространство ядра и пространство пользователя. Эти два понятия охватывают не только два режима исполнения, но так же и то, что каждый из режимов имеет свое собственное отображение памяти – свое собственное адресное пространство.

Unix производит переключение из пространства пользователя в пространство ядра всякий раз, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, исполняющий системный вызов, работает в контексте процесса – от имени вызвавшего процесса и имеет доступ к данным в адресном пространстве процесса. Код, который обрабатывает прерывание, наоборот, являясь асинхронным по своей природе, не относится ни к одному из процессов.

Основное назначение модулей – расширение функциональности ядра. Код модуля исполняется в пространстве ядра. Обычно модуль реализует обе, рассмотренные выше задачи – одни функции выполняются как часть системных вызовов, другие – выполняют обработку прерываний.

Необходимо выделить различия, имеющиеся между модулями ядра и прикладными программами.

Приложение выполняется как цельная задача, от начала и до конца. Модуль же просто регистрирует себя самого в ядре, подготавливая его для обслуживания возможных запросов и его функция "main" завершает свою работу сразу же после вызова. Другими словами, задача функции **init_module** (точка входа) состоит в подготовке функций модуля для последующих вызовов. Вторая точка входа в модуль – **cleanup_module** вызывается непосредственно перед выгрузкой модуля. Возможность выгружать модули – это одна из особенностей модульной архитектуры. С ее помощью вы сможете проверять новые версии модуля без необходимости тратить время на перезагрузку системы.

Рассмотрим конкретный пример создания законченного модуля. Этот модуль не принадлежит ни одному из вышеперечисленных классов. Он называется *skull*, сокращенно от "Simple Kernel Utility for Loading Localities".

Принципы, рассматриваемые при его создании, основаны только на базовых концепциях, а поэтому применимы безотносительно к какому-либо классу устройств.

Простейший драйвер выглядит следующим образом:

```
/*
* skull.c - модуль ядра SKULL
*/
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Student");

/*
* hello_init - функция инициализации, вызывается при загрузке модуля,
* В случае успешной загрузки модуля возвращает значение нуль,
* и ненулевое значение в противном случае.
*/
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

/*
* hello_exit - функция завершения, вызывается при выгрузке модуля.
*/
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Функция hello_init() регистрируется с помощью макроса module_init() в качестве точки входа в модуль. Она вызывается ядром при загрузке модуля. Вызов module_init() – это не вызов функции, а макрос, который устанавливает значение своего параметра в качестве функции инициализации. Все функции инициализации должны соответствовать следующему прототипу.

```
int my_init(void)
```

Так как функция инициализации редко вызывается за пределами модуля, ее обычно не нужно экспортировать и можно объявить с ключевым словом **static**.

Функции инициализации возвращают значение тина **int** . Если инициализация (или то, что делает функция инициализации) прошла успешно, то функция должна возвратить значение нуль. В случае ошибки возвращается ненулевое значение.

В данном случае эта функция просто печатает сообщение и возвращает значение нуль. В настоящих модулях функция инициализации регистрирует ресурсы, выделяет структуры данных и т.д. Даже если рассматриваемый файл будет статически скомпилирован с ядром, то функция инициализации останется и будет вызвана при загрузке ядра.

Функция hello_exit() регистрируется в качестве точки выхода из модуля с помощью макроса module_exit(). Ядро вызывает функцию hello_exit(), когда модуль удаляется из памяти. Завершающая функция должна выполнить очистку ресурсов, гарантировать, что аппаратное обеспечение находится в непротиворечивом состоянии, и т.д. После того как эта функция завершается, модуль выгружается.

Завершающая функция должна соответствовать следующему прототипу.

```
void my_exit(void)
```

Так же как и в случае функции инициализации, ее можно объявить как **static**.

Если этот файл будет статически скомпилирован с образом ядра, то данная функция не будет включена в образ и никогда не будет вызвана (так как если нет модуля, то код никогда не может быть удален из памяти).

Макрос MODULE_LICENSE() позволяет указать лицензию на право копирования модуля. Загрузка в память модуля, для которого лицензия не соответствует GPL, приведет к установке в ядре флага tainted (буквально, испорченное). Этот флаг служит для информационных целей, кроме того, многие разработчики уделяют меньше внимания сообщениям об ошибках, в которых указан этот флаг. Более того, модули, у которых лицензия не соответствует GPL, не могут использовать символы, которые служат "только для GPL" (см. раздел "Экспортируемые символы" ниже в этой главе).

Наконец, макрос MODULE_AUTHOR () позволяет указать автора модуля. Значение этого макроса служит только для информационных целей.

Приложение может вызывать функции, которые не определены в самой программе. На стадии связывания (линковки) разрешаются все внешние ссылки, уходящие во внешние библиотеки. Функция **printf** – одна из таких функций, которая определена в библиотеке libc. Модули так же проходят стадию связывания, но только с ядром, и могут вызывать только те функции, которые экспортируются ядром. Функция **printk** определена в ядре и очень похожа на своего сородича из libc, с небольшими отличиями, главное из которых состоит в отсутствии поддержки вывода чисел с плавающей точкой.

KERN_ALERT определяет уровень вывода сообщений ядра (loglevel). Существуют и другие значения для уровней вывода. Эти макросы раскрываются в строки вида "<x>" (x – уровень вывода), которые объединяются со строкой формата в самом начале сообщения, выводимого функцией printk( ). После этого на основании уровня вывода сообщения и уровня вывода консоли. Т.е. приведенные ниже записи эквивалентны.

```
printk(KERN_ALERT "Goodbye, world\n");
printk("<1> Goodbye, world\n");
```

Поскольку модуль не связывается ни с одной из стандартных библиотек, исходные тексты модуля не должны подключать обычные заголовочные файлы. **В модулях ядра могут использоваться только те функции, которые экспортируются ядром**. Все заголовочные файлы, которые относятся к ядру, расположены в каталогах `include/linux` и `include/asm`, внутри дерева каталогов с исходными текстами ядра (как правило это каталог `/usr/src/linux`).

Исходный код модуля необходимо правильно объединить с деревом исходных кодов ядра. Это можно сделать в виде заплаты или путем добавления в официальное дерево исходного кода ядра. Кроме этого, можно компилировать исходный код модуля отдельно от исходных кодов ядра.

В идеале модуль является частью официального ядра и находится в каталоге исходных кодов ядра. Введение вашей разработки непосредственно в ядро может вначале потребовать больше работы, но обычно такое решение более предпочтительно.

Если вы предпочитаете разрабатывать и поддерживать ваш модуль отдельно от дерева исходных кодов ядра и жить жизнью аутсайдера, просто создайте файл Makefile следующего вида в том каталоге, где находится модуль.

```
obj-m := skull.o
```

Такая конфигурация позволяет скомпилировать файл skull.c в файл skull.ko. (Расширение объектного файла указано как .o, но в результате будет создан модуль с расширением .ko, так это модуль ядра).

Если ваш исходный код занимает несколько файлов, то необходимо добавить две строки.

```
obj-m := skull.o
skull-objs := skull-main.o skull-aux.o
```

Такая конфигурация позволяет скомпилировать файлы skull-main.c и skull-aux.c и создать модуль skull.ko.

Теперь мы создадим Makefile, с помощью которого будем собирать наш модуль. Он будет состоять из одной строчки.

```
obj-m := skull.o
```

Главное отличие от случая, когда модуль находится внутри дерева исходного кода, состоит в процессе сборки. Так как модуль находится за пределами дерева исходных кодов ядра, необходимо указать утилите make местонахождение исходных файлов ядра и файл Makefile ядра. Это также делается просто с помощью следующей команды.

```
make -C/kernel/source/location SUBDIRS=`pwd` modules
```

В этом примере /kernel/source/location – путь к сконфигурированному дереву исходных кодов ядра. Не нужно хранить копию дерева исходных кодов ядра, с которой вы работаете, в каталоге /usr/src/linux, эта копия должна быть где-то в другом месте, скажем где-нибудь в вашем домашнем каталоге.

Теперь разберемся с загрузкой модулей.

Наиболее простой способ загрузки модуля – это воспользоваться утилитой **insmod**. Эта утилита выполняет самые общие действия. Она просто загружает тот модуль, который ей указан в качестве параметра. Утилита insmod не отслеживает зависимости и не выполняет никакой интеллектуальной обработки ошибок. Использовать ее очень просто. От пользователя root необходимо просто выполнить команду

```
insmod module
```

где module – это имя модуля, который необходимо загрузить. Для загрузки нашего модуля необходимо выполнить команду.

```
insmod skull
```

Удалить модуль можно аналогичным образом с помощью утилиты **rmmod**. Для этого от пользователя root нужно просто выполнить команду.

```
rmmod module
```

Например, удалить наш модуль можно следующим образом.

```
rmmod skull
```

Тем не менее, эти утилиты тривиальные и не обладают интеллектуальным поведением. Утилита **modprobe** позволяет обеспечить удовлетворение зависимостей, оповещение об ошибках, интеллектуальную обработку ошибок, а также выполняет множество других расширенных функций. Её настоятельно рекомендуется использовать.

Для загрузки модуля в ядро с помощью утилиты modprobe необходимо от пользователя **root** выполнить команду

```
modprobe module [ module parameters ]
```

где параметр module – это имя модуля, который необходимо загрузить. Все следующие аргументы интерпретируются как параметры, которые передаются модулю при загрузке. Параметры модулей обсуждаются ниже в одноименном разделе.

Утилита modprobe пытается загрузить не только указанный модуль, но и все модули, от которых он зависит. Следовательно, это наиболее предпочтительный механизм загрузки модулей ядра.

Команда modprobe также может использоваться для удаления модулей из ядра. Для этого с правами пользователя root необходимо выполнить ее следующим образом.

```
modprobe -r modules
```

где параметр modules – имя одного или нескольких модулей, которые необходимо удалить. В отличие от команды rmmod, утилита modprobe также удаляет и все модули, от которых указанный модуль зависит, если последние не используются.

Список рекомендуемой литературы

1. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2001. – 736 с.: ил.
2. Харт, Джонсон, М. Системное программирование в среде Win32, 2-е изд.: Пер. с англ.: – М.: Издательский дом «Вильямс», 2001.– 464 с.: ил. – Парал. тит. англ.
3. Лав, Р. Разработка ядра Linux, 2-е изд.: Пер. с англ.: – М.: ООО «И.Д. Вильямс», 2006.– 448 с.: ил. – Парал. тит. англ.
4. Солдатов В.П. Программирование драйверов Windows. Изд. 2-е, перераб. и доп. – М.: ООО "Бином-Пресс", 2004 г. — 480 с: ил.
5. Комиссарова В. Программирование драйверов для Windows. – СПб.: БХВ-Петербург, 2007. – 256 с.: ил.
6. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers, Third Edition – NY: O'Reilly Media, 2005. – 616 с.
7. Гук М.Ю. Аппаратные средства IBM PC. Энциклопедия. 2-е изд. - СПб.: Питер, 2004. – 923 с.: ил.
8. Гук М.Ю. Шины PCI, USB и FireWire. Энциклопедия. - Спб.: Питер, 2005. - 540 с.: ил.
9. Агуров П.В. Интерфейсы USB. Практика использования и программирования. – СПб.: БХВ-Петербург, 2005. – 576 с.: ил.
10. Чан Т. Системное программирование на C++ для Unix: Пер. с англ. – К.: Издательская группа BHV, 1997. – 592 с.
11. Кулаков В. Программирование на аппаратном уровне: специальный справочник (+дискета). 2-е изд. - СПб: Питер, 2003. - 848 с.: ил.
12. Зубков С.В. Assembler для DOS, Windows и UNIX. - М.: ДМК Пресс, 2000. - 608 с.: ил.
13. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.
14. Рудаков П.И., Финогенов К.Г. Программируем на языке ассемблера IBM PC – Изд. 2-е. – Обнинск: Издательство «Питер», 1997. – 584 с., ил.
15. Юров В. Assembler: специальный справочник – СПб: Издательство «Питер», 2000. – 496 с.: ил.
16. Ч. Петзольд. Программирование для Windows 95; в двух томах. Том 1/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 752 с.: ил.
17. Ч. Петзольд. Программирование для Windows 95; в двух томах. Том 2/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 368 с.: ил.
18. У. Моррей, К. Паппас. Создание переносимых приложений для Windows/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 816 с.: ил.

Приложение А
(обязательное)
Форма титульного листа курсового проекта

Министерство образования и науки Российской Федерации

ФГБОУ ВПО Кубанский государственный технологический университет

Кафедра _____ВТ и АСУ_____

Факультет _____

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине _____

(наименование дисциплины)

на тему _____

(тема курсового проекта)

Выполнил(а) студент(ка) группы _____

_____

(Ф.И.О.)

Допущен к защите _____

Руководитель (нормоконтролер) проекта _____

(подпись, дата, расшифровка подписи)

Защищен _____ Оценка _____

(дата)

Члены комиссии _____

_____

_____

(подпись, дата, расшифровка подписи)

Краснодар

20__ г.

Приложение Б
(обязательное)
Форма задания на курсовое проектирование

Министерство образования и науки Российской Федерации

ФГБОУ ВПО Кубанский государственный технологический университет

Кафедра _____
Факультет _____

УТВЕРЖДАЮ
Зав. кафедрой _____
«___» _____ 20___ г.

ЗАДАНИЕ

на курсовое проектирование

Студенту _____ группы _____
факультета _____
специальности _____
_____
(код и наименование специальности)
Тема проекта _____
_____
Содержание задания _____
_____

Объем работы:
а) пояснительная записка _____ стр.
б) программы.

Рекомендуемая литература _____
_____
Срок выполнения: с «___» _____ по «___»_____20__г.
Срок защиты: «___»_____20__г.
Дата выдачи задания: «___»_____20__г.
Дата сдачи проекта на кафедру: «___»_____20__г.
Руководитель проекта _____
(подпись)
Задание принял студент _____
(подпись)

Приложение В
(обязательное)
Пример оформления реферата

Реферат

Пояснительная записка курсовой работы (работы) ___ с., ___ рис., ___табл., __ источников, __ приложений.

ДРАЙВЕР УСТРОЙСТВА, WDM ДРАЙВЕР, DDK (DEVICE DRIVER KIT), DRIVERENTRY, PNP МЕНЕДЖЕР, ТИП ДРАЙВЕРА FDO (FUNCTIONAL DEVICE OBJECT), ФИЛЬТР-ДРАЙВЕРЫ, …

В данной курсовой работе рассмотрены …, произведены …

Объектом исследования является…

Цель работы состоит…

В курсовом проекте освещены вопросы …

К полученным результатам относятся…

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧНИЕ

Методические указания

Составители Мурлин Алексей Георгиевич
Волик Александр Георгиевич

Авторская правка

Редактор
Компьютерная верстка А. Г. Волик

Подписано в печать Формат 60x84/16
Бумага офсетная Офсетная печать
Печ. л. 1,5 Изд. №
Усл. печ. л. 1,4 Тираж 25 экз.
Уч.-изд. л. 1,0 Заказ №
Цена руб.

Кубанский государственный технологический университет
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография КубГТУ: 350058, г. Краснодар, ул. Старокубанская, 88/4