

Московская государственная академия приборостроения и  
информатики

Кафедра "Персональные ЭВМ"

**А. В. Рощин**

**ОРГАНИЗАЦИЯ ВВОДА–ВЫВОДА**

Часть 1

**ВИРТУАЛЬНЫЕ ДРАЙВЕРЫ  
И ВИРТУАЛЬНОЕ ОКРУЖЕНИЕ WINDOWS**

Москва 2002

УДК 681.3

Организация ввода-вывода. Часть 1. Виртуальные драйверы и виртуальное окружение WINDOWS. Учебное пособие/ А.В.Рощин. – М.: МГАПИ, 2002. – 82 с.: ил.

ISBN 5-8068-0248-5

Рекомендовано Ученым Советом МГАПИ в качестве учебного пособия для специальности 2201.

Рецензенты: профессор Зеленко Г.В.  
доцент Туманов М.П.

Предлагаемая работа может рассматриваться как пособие-справочник для студентов, осваивающих основы системного программирования под WINDOWS 95/98 (в дальнейшем весь клон операционных систем WINDOWS 95/08 будет называться WINDOWS 95). В пособии рассматриваются вопросы организации работы приложений, как с устройствами ввода-вывода, так и с аппаратурой ЭВМ в целом. Достаточно подробно рассматриваются вопросы организации виртуального окружения приложения, как основы взаимодействия аппаратной и программной частей вычислительной системы.

Для успешной работы с этим пособием необходимо знание основ написания драйверов устройств под DOS или другой операционной системой. Необходимы также знания языка Си и ассемблера 80x86.

Л  $\frac{1402010000}{ЛР020418-97}$  без объявл.

ISBN 5-8068-0248-5

© А.В.Рощин. 2002.

	ВВЕДЕНИЕ	5
1.	ОКРУЖЕНИЕ WINDOWS И ВИРТУАЛЬНЫЕ ДРАЙВЕРЫ	7
1.1.	Виртуальный мир Windows	7
1.2.	Что такое виртуальная машина?	8
1.3.	Режимы работы процессора	9
1.4.	Защищенный режим	10
1.5.	Режим V86	12
1.6.	Исполнительное окружение Windows	13
1.7.	Резюме	16
1.8.	Контрольные вопросы	17
2.	КАК WINDOWS РЕАЛИЗУЕТ ВИРТУАЛЬНОЕ ОКРУЖЕНИЕ	18
2.1.	Захват доступа к портам ввода-вывода	18
2.2.	Перехват обращений к устройствам, размещенным в адресном пространстве памяти	21
2.3.	Перехват прерываний и исключений	22
2.4.	Регистры процессора	24
2.5.	Удовлетворение запросов адресов Win32, Win16 и приложений DOS	24
2.6.	Резюме	28
2.7.	Контрольные вопросы	29
3.	ВВЕДЕНИЕ В VxD	31
3.1.	Загрузка VxD	32
3.2.	Базовая структура VxD	33
3.3.	Блок дескриптора устройства	36
3.4.	Поддержка структур данных	39
3.5.	Уведомление о событии	45
3.6.	Сообщения инициализации и завершения статически загружаемого VxD	48
3.7.	Сообщения инициализации и завершения динамически загружаемого VxD	50
3.8.	Сообщения об изменении состояния VM	51
3.9.	Потоковые сообщения	53
3.10.	Отличие от Windows 3.x	55
3.11.	Резюме	55
3.12.	Контрольные вопросы	56

4.	СТРУКТУРА VxD	59
4.1.	Инструменты для создания VxD	59
4.2.	Исходные файлы версии “ DDK ”	62
4.3.	Блок дескриптора устройства DDB и управляющая процедура устройства: SKELCTRL.ASM	67
4.4.	Резюме	79
4.5.	Контрольные вопросы	79
	Литература	81

## **ВВЕДЕНИЕ**

Операционная система Windows 95 предоставляет приложению очень сложное окружение. Цель этого пособия заключается в том, чтобы помочь студенту понять, что именно из этого окружения является существенным для различных типов драйверов.

В самом широком определении – "драйвер" это набор функций, управляющих устройством. Один путь категоризации драйверов – это способ упаковки функций. В мире DOS драйвером может быть как модуль, который физически находится в приложении, так и модуль, не имеющий к приложению никакого отношения (собственно драйвер устройства DOS или резидентная программа). В мире Windows, драйвером может быть модуль, который динамически связан с приложением (называемый DLL), или модуль, не связанный с приложением (называемый виртуальным драйвером – VxD).

Другой путь категоризации драйверов – привилегии. Некоторые операционные системы, типа UNIX и Windows NT, отстраняют приложения от непосредственного управления аппаратными средствами. В этих средах, только привилегированным частям кода, известным как "драйверы устройств", позволяют иметь дело с аппаратными средствами. Приложения, которые должны управлять аппаратными средствами, должны пользоваться услугами этих драйверов.

Windows 95/98 также поддерживает привилегированные драйверы. В Windows, эти драйверы устройств называются VxD – виртуальные драйверы. Однако, Windows 95/98 позволяет работать с аппаратными средствами и посредством DLL. В Windows 95/98 DLL-интерфейс с аппаратными средствами также часто называется драйвером.

Третий путь категоризации драйверов – интерфейс, который драйвер представляет приложению и ядру операционной системы. Все драйверы

Windows NT используют единообразный интерфейс с ядром NT. Ядро в свою очередь обеспечивает стандартный интерфейс для вызова любого драйвера. Привилегированный драйвер в Windows 95/98 отличается от драйвера NT. Хотя виртуальные драйверы Windows 95/98 использует тот же интерфейс ядра системы, нет никакого стандартного интерфейса между VxD и прикладным уровнем. Вместо этого, каждый VxD определяет свой собственный прикладной интерфейс.

## 1. ОКРУЖЕНИЕ WINDOWS И ВИРТУАЛЬНЫЕ ДРАЙВЕРЫ

### 1.1. Виртуальный мир Windows

Windows 95 управляют тремя различными типами приложений: приложения DOS, приложения Win16 и приложения Win32. Для преодоления потенциальной несовместимости различных типов приложений Windows выполняет их на виртуальных машинах в виртуальном окружении. При разработке приложений под Windows программисты могут обычно игнорировать различие между виртуальным и реальным окружением. Для большинства приложений, виртуальное окружение совпадает с реальным.

Это, однако, не относится к написанию виртуальных драйверов VxD, потому что VxD выполняется в контексте супервайзера, который выполняется вне любой из виртуальных машин. Фактически, VxD становится частью программного обеспечения, которое реализует виртуальную машину. Следовательно, при написании драйвера VxD необходимо полное понимание того, чем виртуальное окружение отличается от физического, и как Windows создает иллюзию виртуальной машины. Полное понимание виртуальной машины особенно важно для программистов, разрабатывающих драйверы VxD, которые должны управлять ресурсами в виртуальном окружении приложения, что обычно и имеет место.

В этой главе рассмотрены существенные аспекты архитектуры Windows, включая механизм осуществления виртуальной машины, основные характеристики виртуального окружения, и характеристики окружения супервайзера.

## 1.2. Что такое - виртуальная машина?

Виртуальная машина – это создаваемая системой иллюзия аппаратных и программных средств для приложения. Виртуальные ресурсы – это эмуляции аппаратных средств ЭВМ (иногда и программного обеспечения). Для реализации виртуальных ресурсов, эмуляция должна быть настолько полна, чтобы для написания типовой программы было все равно, реальны аппаратные средства ЭВМ, или эмулированы. Например, система виртуальной памяти использует место на жестком диске, системное программное обеспечение, специальные средства процессора, и относительно небольшой объем физической оперативной памяти, чтобы эмулировать систему с неограниченным объемом физической памяти. Эмуляция настолько убедительна, что программы, запускаемые в виртуальной среде, могут быть написаны так, как если бы полное виртуальное адресное пространство совпадало с фактически имеющейся физической памятью. Про такую систему памяти говорят, что она **виртуализована**.

Когда система виртуализует все, или почти все, доступные для приложения ресурсы, образуется "виртуальная машина", или VM. Доступные для программы ресурсы включают регистры процессора, память и периферийные устройства (дисплей, клавиатура, и т.д.). Реальная причина использования виртуальных машин под Windows состоит в том, чтобы поддержать существующие приложения DOS. Приложение DOS считает, что оно единственное запущенное приложение, что оно имеет непосредственный доступ к аппаратным средствам ЭВМ, использует всю доступную память системы, и полностью использует время процессора. С тех пор, как под Windows приложение DOS не является единственным запущенным приложением, Windows создает виртуальную машину для запуска приложения. Обращения к аппаратным средствам

перехватываются и могут быть перенаправлены. Вместо оперативной памяти может быть использовано место на жестком диске, и виртуальная машина может быть "погружена в спячку" в то время, как ресурсы процессора получает другая виртуальная машина.

Понятие виртуальной машины может быть сформулировано так: задача с собственным исполнительным окружением, которая включает собственные

- адресное пространство,
- пространство портов ввода - вывода,
- прерывания, и
- регистры процессора.

Главный процесс супервайзера, называемый *менеджером виртуальных машин* (VMM), основан на способности аппаратных средств ЭВМ создавать не только одну виртуальную машину, но и несколько независимых виртуальных машин с собственным виртуальным исполнительным окружением. Все приложения Windows (и Win32, и Win16) управляются отдельной VM, называемой системной VM, притом, что каждое приложение DOS выполняется собственной независимой VM. Каждое из этих виртуальных окружений может существенно отличаться от реальных аппаратных средств.

### 1.3. Режимы работы процессора

Чтобы создавать и обслуживать виртуальные машины, VMM использует специальные свойства семейства процессоров 80x86. Эти процессоры могут работать в любом из трех режимов: защищенный, реальный, и V86. Windows 95 используют два режима: защищенный режим и режим V86.

Режим работы процессора определяет некоторые важные характеристики выполнения:

- сколько памяти может адресовать процессор,
- как процессор транслирует логические адреса, формируемые программой, в физические адреса на шине,
- как процессор защищает доступ к памяти и портам ввода-вывода и предотвращает выполнение некоторых инструкций.

Для Windows 95 необходим процессор 80386 или выше. В дальнейшем изложении когда, используется термин *процессор*, имеется в виду именно такой процессор. Будут использоваться термины "32-разрядный защищенный режим" и "16-разрядный защищенный режим", относящиеся к процессору, работающему в защищенном режиме и выполняющему 32- или 16-разрядное приложение соответственно. Хотя технически они не являются "режимами" в том смысле, в каком ими являются V86 и защищенный (то есть они не определяются битами в регистре флагов), разрядность выполняемого кода так влияет на поведение процессора, что 32-разрядный защищенный режим должен рассматриваться иначе, чем 16-разрядный защищенный режим.

#### **1.4. Защищенный режим**

Самое большое различие между 32-разрядным и 16-разрядными защищенными режимами – количество адресуемой памяти. В 16-разрядном защищенном режиме адресуется лишь 16Мб. В 32-разрядном защищенном режиме процессор может адресовать 4Гб ( $2^{32}$ ). Обычно в системе не бывает столько физической памяти (4Гб). Для реализации такого большого адресного пространства обычно используется виртуальная память.

Хотя это различие в размере адресуемой памяти конечно важно, более важным является различие в размере сегмента — максимального объема памяти, адресуемого непосредственно. В 16-разрядном защищенном режиме размер сегмента ограничен 64Кб ( $2^{16}$ ), и разработчики больших программ должны помнить о сегментах. В 32-разрядном защищенном режиме размер сегмента может достигать 4Гб — настолько большого, что большая часть операционных систем, использующих 32-разрядный защищенный, включая Windows 95, делают сегментацию невидимой для программиста, создавая единственный сегмент, который адресует все 4Гб. В этом случае приложению нет необходимости менять сегменты.

В Windows 95 и 32- и 16-разрядный защищенные режимы используют один и тот же способ трансляции логических адресов в физические, выдаваемые на шину. Трансляция адресов производится в два этапа. Логический адрес, состоящий из селектора и смещения, переводится сначала в промежуточную форму, называемую линейным адресом, путем поиска по селектору в таблице дескрипторов, которая содержит базовый линейный адрес сегмента. Второй этап трансляции — трансляция линейного адреса в физический адрес — называется формированием страниц. Подробно этот двухэтапный процесс трансляции будет объяснен позже. Пока надо лишь помнить, что на первом шаге используется просмотр селектора, для нахождения линейного адреса, что не совпадает с первым шагом режима V86.

Термин "защищенный режим" возник потому, что это был первый режим процессора 80x86, обеспечивающий механизм управления доступом к памяти и портам ввода-вывода — механизм, который операционная система могла использовать для защиты себя от приложений. Весь этот механизм основан на концепции уровня

привилегий. Выполняемый код всегда имеет определенный уровень привилегий, который на жаргоне фирмы Intel называется "кольцо", причем кольцо 0 – самое внутреннее и наиболее привилегированное кольцо, а кольцо 3 наиболее внешнее и наименее привилегированное.

Уровень привилегий сегмента кода определяется операционной системой, при этом определяется, к каким областям памяти и портам ввода-вывода может иметь доступ этот код, а также какие инструкции в нем могут выполняться. Код кольца 0, который ранее был назван кодом супервайзера, может иметь доступ к любой области памяти или любому порту ввода-вывода и может выполнять любые инструкции. Если приложение, выполняющееся во внешнем кольце, пытается сделать что-то, что не позволяет делать его уровень привилегий, процессор вырабатывает исключение.

### *1.5. Режим V86*

Если учесть, что защищенный режим был создан для поддержки больших программ и более совершенных операционных систем, режим виртуальной реальной машины V86 используется для эмуляции реального режима – единственного режима, поддерживаемого первоначальными PC, и единственного режима, поддерживаемого приложениями DOS даже сегодня. Режим V86 имеет то же максимальное адресное пространство 1Мб, что и реальный режим. Трансляция адресов в режиме V86 связывает реальный и защищенный режимы. Режим V86 преобразует логический адрес в линейный так же, как это делается в реальном режиме – сегмент просто сдвигается влево на 4 разряда. (Сравните это с поиском по селектору, используемым в защищенном режиме.) Преобразование линейного адреса в физический в режиме V86 производится так же, как в

защищенном режиме – методом страничного преобразования. Страничное преобразование полностью прозрачно для приложений DOS .

Для предохранения от крушения системы множественными приложениями DOS, режим V86 использует те же механизмы защиты, что и защищенный режим. Любая программа, выполняемая в режиме V86, инициирует исключение (передачу управления операционной системе) если делается попытка выполнить некоторые "привилегированные" инструкции, доступ к некоторым портам ввода-вывода или доступ к запрещенным областям памяти. В таблице 1.1 приведены параметры исполнительного окружения для процессоров 80386+.

Таблица 1.1

Варианты физического исполнительного окружения для различных режимов процессора 80386+

	Защищенный 32-разрядный	Защищенный 16-разрядный	V86
Полное адресное пространство	4 Гб ( $2^{32}$ )	16 Мб ( $2^{24}$ )	1Мб ( $2^{20}$ )
Размер сегмента	4Гб	64Кб	64Кб
Трансляция адресов	Логический в линейный: поиск по селектору Линейный в физический: таблицы страниц	Логический в линейный: поиск по селектору Линейный в физический: таблицы страниц	Логический в линейный: сдвиг сегмента влево на 4 разряда Линейный в физический: таблицы страниц
Уровень привилегий	0 - 3	0 - 3	3
Механизм защиты	есть	есть	есть

### 1.6. Исполнительное окружение Windows

Архитектура Windows 95 поддерживают четыре существенно различных типа процессов: процессы супервайзера, приложения Win32, приложения Win16 и приложения DOS. Windows 95 управляют каждым из них в различном исполнительном окружении. Исполнительное окружение

может быть описано режимом процессора, уровнем привилегий, и разрядностью (16 или 32). В таблице 1.2 систематизированы варианты исполнительного окружения Windows.

Процессы супервайзера работают в кольце привилегий 0 защищенного режима (самый высокий уровень доступа), так что они способны видеть и управлять фактическим аппаратным окружением ЭВМ. Процессы супервайзера выполняются на текущей машине, а не на виртуальной или, другими словами, процессы супервайзера выполняются вне любой виртуальной машины. Из всех компонентов, которые составляют Windows 95, только VMM (менеджер виртуальных машин) и VxD выполняются в окружении супервайзера. Все другие компоненты работают в виртуальной машине.

Табл. 1.2

Исполнительное окружение Windows, связанное с различными типами процессов

Тип процесса	Режим процессора	Привилегии	Разрядность	Модель памяти	Виртуальная машина
Супервайзер	защищенный	Кольцо 0	32	плоская	ни в одной
Win32	защищенный	Кольцо 3	32	плоская	Системная
Win16	защищенный	Кольцо 3	16	сегментированная	Системная
DOS	V86	Кольцо 3	16	сегментированная	Индивидуальная

Окружение супервайзера 32-разрядное, так что эти процессы могут адресовать 4Гб виртуальной памяти. Процессы супервайзера используют только два селектора, каждый из которых адресует 4Гб. Эти два селектора отличаются только атрибутами: один помечен как выполняемый и загружен в CS, другой помечен невыполняемым и загружен в DS, ES, и SS. (Эти атрибуты селектора хранятся в той же таблице дескрипторов, который хранит базовый линейный адрес сегмента.) Этот тип модели

памяти, где сегменты загружены раз и навсегда, называется плоской моделью, и делает сегментацию по сути невидимой для программиста.

В то время, как процессы супервайзера работают вне любой виртуальной машины (на реальной машине), процессы Win32 работают в кольце 3 (самая низкая привилегия доступа) в виртуальной машине. Кроме того, все процессы Win32 работают в одной и той же виртуальной машине, называемой системной виртуальной машиной. Процессы Win32 являются процессами защищенного 32-разрядного режима, использующими плоскую модель памяти, подобно процессам супервайзера. Непосредственно адресуя 4Гб памяти, они могут практически всегда игнорировать селекторы и сегменты.

Процессы Win16 работают в той же самой системной виртуальной машине, что и процессы Win32. Процессы Win16 работают в кольце привилегий 3 защищенного режима, но не имеют преимуществ плоской модели памяти. Поскольку они выполняются в 16-разрядном защищенном режиме, процессы Win16 все еще увязли в 16-мегабайтном адресном пространстве и должны иметь дело с селекторами и 64-килобайтными сегментами.

Каждый процесс DOS получает свою собственную виртуальную машину. Процесс DOS не работает в защищенном режиме подобно всем другим типам процессов. Вместо этого он выполняется в режиме V86, эмулирующим в процессоре 80386 процессор 8086. Режим V86 означает сегментированную модель памяти с трансляцией типа 8086 плюс дополнение страничной организации памяти. Режим V86 также подразумевает кольцо 3 привилегий, так что доступ к аппаратным ресурсам ЭВМ и прерываниям скрыт и виртуализован.

Почему каждый процесс DOS получает свою собственную виртуальную машину, в то время как все приложения Win32 и Win16

разделяют системную виртуальную машину? Поскольку процессы DOS в принципе не понимают, что они разделяют систему с другими процессами, они обычно "занимают" машину полностью. Для процессов DOS типично изменение таблицы векторов прерываний и непосредственный вывод на экран. Windows управляют каждой программой DOS в отдельной виртуальной машине так, чтобы каждая программа изменяла бы только свою собственную виртуальную таблицу векторов прерываний, и осуществляла бы вывод на собственный виртуальный экран.

В отличие от приложений DOS, Windows приложения (и Win32 и Win16), знают, что запущены другие процессы. Они осуществляют вывод только в собственные окна, а не непосредственно на экран, и используют вызов операционной системы для изменения таблицы векторов прерываний вместо непосредственного ее изменения. Windows-приложения не требуют такой защиты друг от друга, как от приложений DOS, которые не знают о других приложениях. Так что Windows могут успешно управлять всеми Windows-приложениями в одной и той же виртуальной машине.

### *1.7. Резюме*

Windows может запускать Win32, Win16 и приложения DOS, в том числе и в многозадачном режиме. Windows делает это, запуская приложения не на реальной машине, а на виртуальных машинах. Менеджер виртуальных машин (являющийся процессом супервайзера), выполняется на реальной машине и обеспечивает каждый из различных типов приложений с различным виртуальным окружением.

### 1.8. Контрольные вопросы

1. Зачем нужны виртуальные машины?
2. Что такое виртуальные машины?
3. Что такое супервайзер?
4. Что такое менеджер виртуальных машин?
5. Что такое кольца привилегий?
6. Чем отличаются приложения Win32, Win16 и DOS?
7. Где выполняются приложения Win32?
8. Где выполняются приложения Win16?
9. Где выполняются приложения DOS?
10. Почему приложения Win32 могут выполняться в одной виртуальной машине?
11. Почему приложения DOS не могут выполняться в одной виртуальной машине?
12. Что такое системная виртуальная машина?
13. Что такое плоская модель памяти?
14. Что такое сегментированная модель памяти?
15. Как формируется физический адрес в защищенном режиме?
16. Как формируется физический адрес в режиме V86?
17. В каких режимах может работать процессор 80386?
18. Что такое исполнительное окружение Windows?
19. Какие типы процессов реализуются в Windows 95/98?
20. Что такое режим Win32?
21. Что такое режим Win16?
22. Что такое привилегированные инструкции?
23. Что такое виртуализация аппаратных ресурсов?

## 2. КАК WINDOWS РЕАЛИЗУЕТ ВИРТУАЛЬНОЕ ОКРУЖЕНИЕ

В предыдущей главе рассмотрена концепция виртуальной машины и четырех ее составляющих: пространства ввода-вывода, операций прерывания, регистров процессора, и адресного пространства. В ней были описаны также различные типы виртуального окружения различных типов процессов, которые выполняются под Windows: Win32, Win16, DOS, и супервайзер (VMM и VxD). Теперь более подробно рассмотрим, как менеджер виртуальных машин виртуализует каждый из компонентов виртуальной машины для каждого типа процесса. (Эта глава предполагает знакомство читателя с основными особенностями архитектуры Intel 80x86.)

### 2.1. Захват доступа к портам ввода - вывода

И защищенный режим, и режим V86 позволяют операционной системе перехватывать инструкции ввода и вывода, и, таким образом, предотвращать непосредственное обращение приложения устройствам, расположенным в пространстве ввода-вывода. К устройствам, расположенным в адресном пространстве памяти, обращаются при помощи любой инструкции, использующей обращение к памяти, в то время как к устройствам, расположенным в пространстве ввода-вывода, обращаются только при помощи инструкций ввода и вывода. Windows 95 использует комбинацию двух возможностей управления доступом к адресам ввода-вывода – **Уровень привилегий ввода-вывода (IOPL)** и **Карту разрешения ввода-вывода (IOPM)**.

В защищенном режиме, каждый сегмент кода имеет связанный **Уровень привилегий дескриптора**, хранящийся в таблице дескрипторов. Каждый сегмент кода имеет также отдельный атрибут для **Уровня**

**привилегий ввода-вывода (IOPL)**, также хранящегося в таблице дескрипторов. Когда инструкция ввода или вывода выполняется в защищенном режиме, процессор сравнивает IOPL сегмента с текущим уровнем привилегий выполняемого сегмента кода (называемый *CPL-current privilege level*, текущий уровень привилегий). Если  $CPL < IOPL$ , сегмент имеет достаточно привилегий, и процессор выполняет инструкцию. Если  $CPL \geq IOPL$ , процессор использует IOPM как второй уровень защиты. IOPM – побитовая карта портов: бит 1 означает "нет доступа", бит 0 – "доступ разрешен". Так, если  $CPL \geq IOPL$  и бит IOPM соответствующего порта сброшен, инструкция выполняется. Но если бит IOPM этого порта установлен, процессор генерирует соответствующее исключение.

Для Windows 95 IOPM – доминирующий механизм определения привилегий для всех виртуальных машин. В виртуальных машинах DOS, IOPM определяет привилегии ввода-вывода приложения, потому что менеджер виртуальных машин VMM управляет приложениями DOS в режиме V86, где процессор игнорирует IOPL и учитывает только IOPM при обработке инструкций ввода и вывода. В виртуальных машинах Win16 и Win32, IOPM определяет привилегии ввода-вывода приложения, потому что менеджер виртуальных машин VMM управляет всеми процессами Win16 и Win32 с  $CPL > IOPL$ . Хотя приложения Win16 и Win32 работают в защищенном режиме, где процессор проверяет IOPL, эта проверка всегда кончается последующей проверкой IOPM.

Управляя IOPM, Windows 95 может перехватывать доступ к определенным портам при разрешении свободного доступа к другим портам. Windows 95 использует эту способность для виртуализации физических устройств, расположенных по адресам, которые следует перехватить. При обработке доступа к устройству через виртуальный

драйвер устройства (VxD), Windows 95 может обслуживать раздельную информацию о состоянии для каждой из виртуальных машин, которая хотела бы использовать устройство.

Менеджер виртуальных машин VMM ответствен за поддержание IOPM. Виртуальный драйвер (VxD) вызывает сервис VMM, чтобы запросить перехват соответствующего порта. При создании этого запроса, VxD задает функцию отзыва, называемую "обработчик перехвата портов (*port trap handler*)". Менеджер виртуальных машин VMM отвечает на такой запрос установкой бита этого порта в IOPM. Когда виртуальная машина обращается к этому порту, и таким образом вызывает ошибку, обработчик ошибок менеджера виртуальных машин VMM вызывает зарегистрированный в VxD обработчик перехвата портов. Этот обработчик перехвата портов может сделать что-нибудь в ответ на попытку ввода-вывода: VxD может игнорировать инструкцию, выполнить ее, или может заменить значение (например. OUT 3F8h, 01h мог бы стать OUT 3F8h, 81h).

Windows 95 и его стандартный компонент VxD перехватывает почти все стандартные PC устройства ввода-вывода, но никогда не перехватывают нестандартные адреса ввода-вывода. В таблице 2.1 приведен перечень перехваченных портов. Нестандартные виртуальные драйверы (VxD) могут перехватывать и другие порты.

Таблица 3.1

Захват портов ввода-вывода стандартными виртуальными драйверами (VxD) для Windows 95

Адрес порта	VxD	Описание
3F0 - 3F2, 3F4, 3F5, 3F7	VFBACKUP	Контроллер гибких дисков
1F0 - 1F7	ESDI_506	Контроллер жесткого диска
378, 379, 37A	VPD	Принтер LPT1
2F8 - 2FE, 3F8 - 3FE	SERIAL	Последовательные порты COM1 и COM2
61	VSD	Звук
3B4, 3B5, 3BA, 3D0 - 3DF, 3C0 - 3CF	VDD	VGA дисплей
1CE, 1CF, 2E8, x6EC-EF, AEC-EF, xEEC-EF	ATI	Минипорт дисплея PCI VGA
00-0F, C0-DF, 81-83, 87, 89, 8A	VDMAD	Контроллер прямого доступа
60, 64	VKD	Клавиатура
40, 43	VTD	Таймер
20, 21, A0, A1	VPICD	Программируемый контроллер прерываний

## *2.2. Перехват обращений к устройствам, размещенным в адресном пространстве памяти*

В то время, как большинство стандартных внешних устройств размещается в адресном пространстве ввода-вывода, некоторые размещены в адресном пространстве памяти. Windows 95 полагается, прежде всего, на механизм отказа страницы при виртуализованном доступе к устройствам, размещенным в адресном пространстве памяти. Чтобы перехватить обращение к одному из этих устройств, виртуальный драйвер (VxD) виртуализованного устройства пометит страницу, соответствующую физическому адресу устройства, как "не представленная", и регистрирует собственный обработчик отказов страниц при помощи менеджера виртуальных машин VMM. Если процесс, выполняющийся в виртуальной машине, пытается получить доступ к этой странице, это вызовет отказ страницы. Вместо стандартного ответа и

попытки "подкачать" страницу, обработчик ошибок VMM теперь обратится к зарегистрированному обработчику отказа страницы в виртуальном драйвере (VxD), который виртуализует данное устройство. Тогда обработчик VxD может решить, какое действие соответствует требованиям виртуального окружения.

Виртуальный драйвер дисплея (VDD) использует этот механизм для виртуализации буфера видеостраницы. Когда программа DOS выполняет запись в видеобуфер по логическому адресу b000:0000, информация не появляется на экране, потому что VDD пометил эту конкретную страницу как "не представленная". Вместо этого, обращение к видеостранице перехватывается обработчиком отказа страниц виртуального драйвера (VxD) и переадресуется в другую область физической памяти. Это объясняет, каким образом видеоинформация выводится в окно, вместо того, чтобы появиться на полном экране. VxD использует тот же механизм для арбитража доступа к другому устройству, размещенному в адресном пространстве памяти – монохромному адаптеру.

### *2.3. Перехват прерываний и исключений*

Кроме перехвата обращений к памяти и устройствам ввода-вывода, Windows 95 перехватывает некоторые "привилегированные" инструкции. К "привилегированным" инструкциям относятся инструкции, которые могут использоваться для обхода механизмов защиты процессора, или могут нарушить целостность виртуальной машины. К привилегированным инструкциям относятся те инструкции, которые воздействуют на флаг разрешения прерывания процессора (CLI, STI, POPF, RET), программные прерывания (INT n), а также инструкции загрузки таблиц дескрипторов (LLDT, LDGT, LIDT). В основном Windows 95 перехватывает эти инструкции для защиты целостности виртуальных машин. В случае

инструкции INT n Windows 95 использует захват для обеспечения прозрачности перехвата запросов BIOS и DOS.

Процессы, запущенные в виртуальной машине выполняются с привилегиями кольца 3 (наименьшие привилегии). Код, выполняемый в кольце 3, вызывает генерацию исключений при выполнении любой из этих "привилегированных" инструкций. Когда это исключение сгенерировано, процессор переключается в кольцо 0 и затем передает управление соответствующему обработчику.

Более точно, каждый сегмент имеет связанный с ним *Дескрипторный уровень привилегий* (DPL). Этот сегментный уровень привилегий определяет уровень привилегий большинства инструкций (например, LLDT, LGDT). Однако, некоторые инструкции (те, которые воздействуют на флаг разрешения прерываний процессора) получают свой уровень привилегий из IOPL, а не из DPL. Если, например, процесс в кольце 3 выполняет инструкцию STI или CLI, процессор генерирует исключение, только если  $CPL > IOPL$ .

Более существенные различия между окружением системной виртуальной машины и окружением виртуальной машины DOS касаются этих привилегий, обусловленных IOPL. В то время, как архитектура 80386 поддерживает перехват CLI и STI в обоих защищенных режимах и в режиме V86, Windows 95 не перехватывает инструкции STI и CLI в режиме V86. Менеджер виртуальных машин VMM преднамеренно устанавливает  $CPL = IOPL$  для приложения DOS, так что инструкции CLI и STI не вызывают исключения. Более того, хотя Windows 95 управляют приложениями Win16 и Win32 с  $CPL > IOPL$  так, чтобы CLI/STI вызывали исключения для приложений Windows, обработчик исключений менеджера виртуальных машин идет вперед и выполняет инструкцию, позволяя или запрещая прерывание от имени приложения. Очевидно,

разработчики решили, что накладные расходы на перехват инструкций STI и CLI были бы большей платой за быстроедействие, чем они могли бы заплатить.

#### *2.4. Регистры процессора*

Виртуализация третьего ресурса – регистров процессора – тривиальна по сравнению с механизмами виртуализации пространства портов ввода-вывода и прерываний. Менеджер виртуальных машин (VMM) поддерживает структуру данных виртуальных регистров для каждой виртуальной машины. Каждый раз при переключении от выполнения одной виртуальной машины (например, VM1) к выполнению другой виртуальной машины (например, VM2), менеджер виртуальных машин сначала сохраняет состояние регистров VM1 в виртуальной структуре регистров VM1, затем, перед выполнением VM2 изменяет текущее состояние регистров процессор из виртуальной структуры регистров VM2.

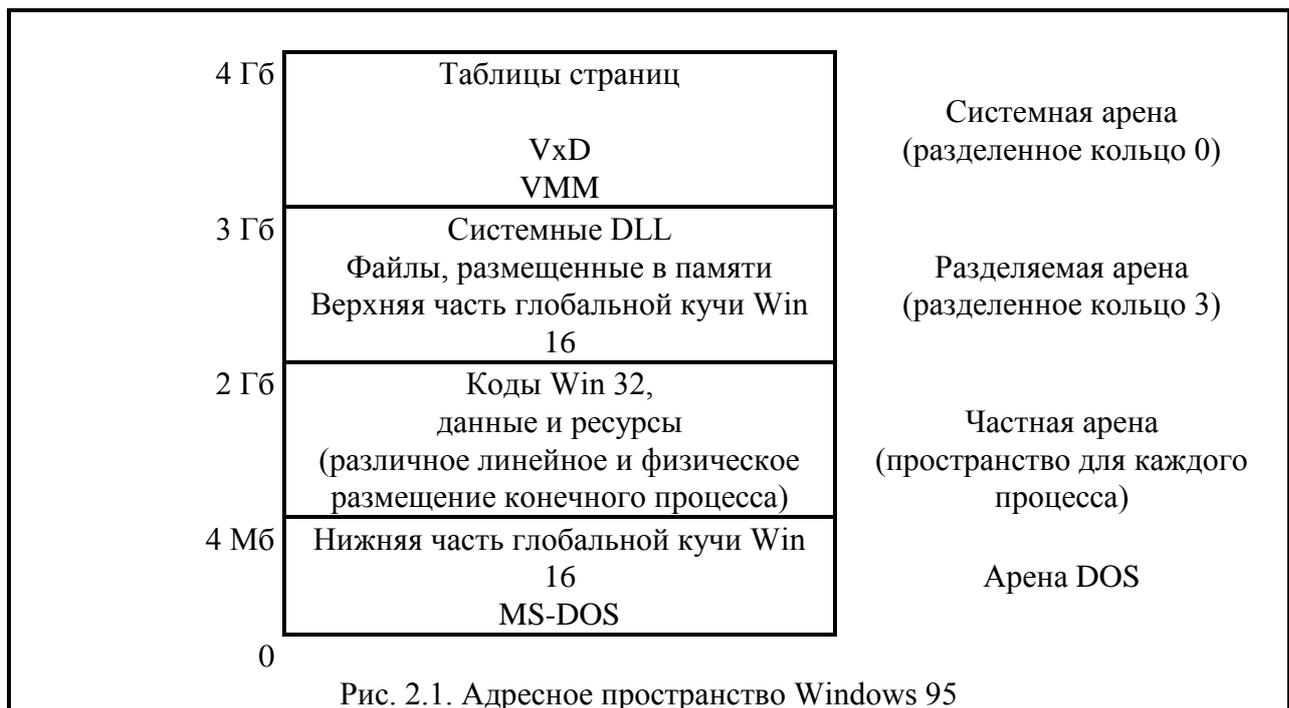
#### *2.5. Удовлетворение запросов адресов Win32, Win16, и приложений DOS*

В отличие от Windows 3.x, использующих малую часть 4Гб линейного адресного пространства, Windows 95 использует его целиком. Windows 95 делит эти 4Гб на несколько областей, называемых аренами (рис. 2.1):

- частная арена,
- разделяемая арена,
- системная арена, и
- арена DOS.

Частная арена, от 4Мб до 2Гб (почти половина полного 4Гб пространства) используется для кода, данных и ресурсов приложений

Win32. Эта арена частная, поэтому она различно расположена в физической памяти для каждого приложения Win32. Так, например, когда приложение\_1 Win32 обращается к 4-мегабайтной области линейных адресов, оно получает в распоряжение одну физическую область, в то время, как приложение\_2 Win32, обращаясь к той же 4-мегабайтной области линейных адресов, получает в распоряжение другую физическую область. Windows 95 достигают этого эффекта, переключая 511 входов каталога страниц (*page directory entries*) этого линейного пространства от 4Мб до 2Гб. При выполнении приложения\_1 Win32, этот вход каталога страниц указывает на один набор таблиц страниц (*page tables*) (рис. 2.2). Если Windows 95 переключается на выполнение приложения\_2 Win32, они указывают на другой набор таблиц страниц (рис. 2.3).



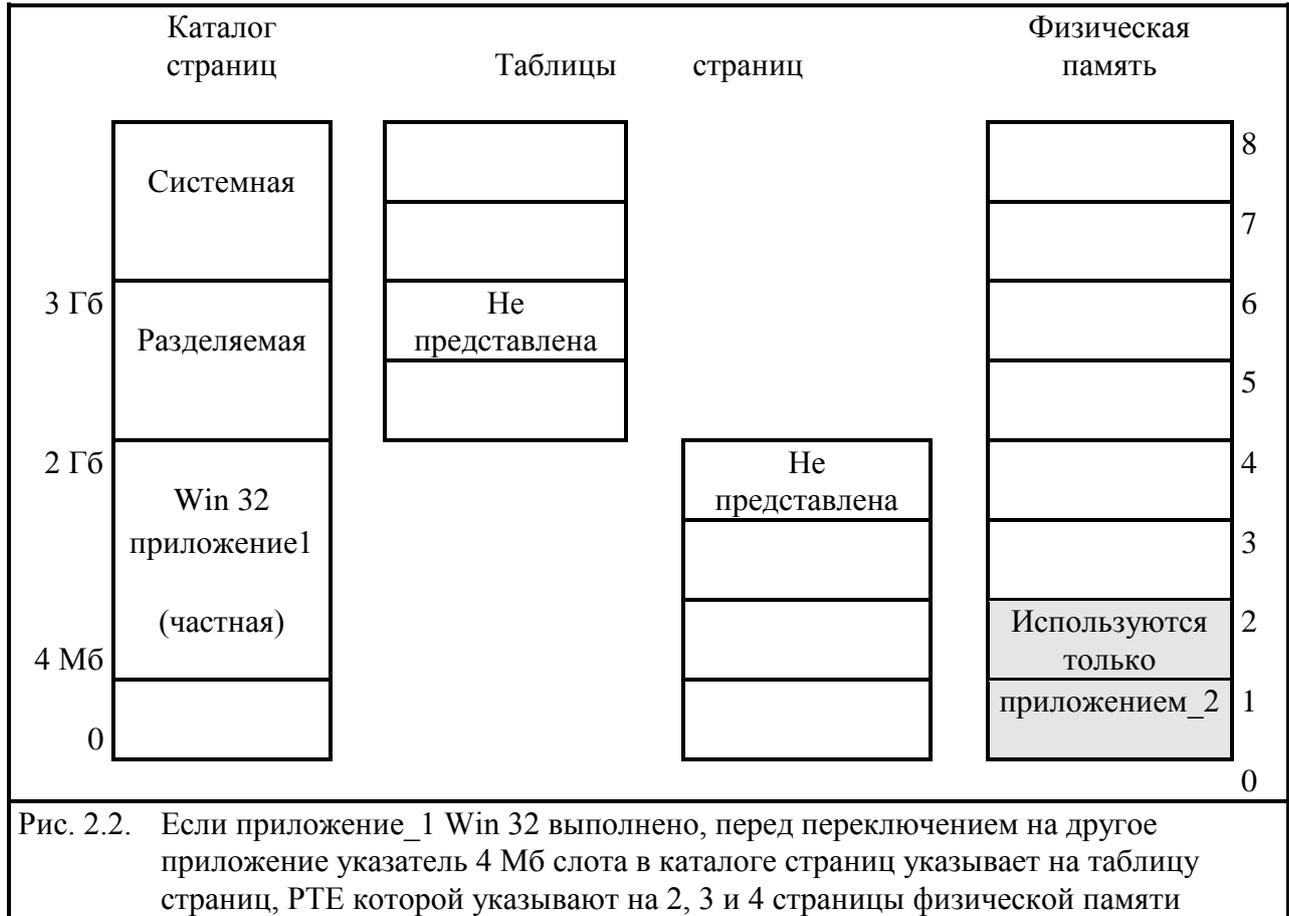
Изменяя входы каталога страниц (PDE) частной arenas, Windows 95 защищают приложения Win32 друг от друга. Входы таблиц страниц (PTE), используемые для приложения\_1 Win32 просто не содержат физических адресов, используемых приложением\_2, а входы таблицы страниц,

используемых приложением\_2 Win32, не содержат физических адресов, используемых приложением\_1. Приложение\_1 и приложение\_2 просто не могут навредить ресурсам друг друга.

Многие операционные системы предотвращают непосредственное обращение к системным страницам компонентов пользователя. Для этого для системных страниц в РТЕ устанавливается бит супервайзера, который вызывает отказ страницы, если к системной странице происходит обращение из режима пользователя.

Windows 95 вообще не используют биты супервайзера, что упрощает передачу данных между виртуальным драйвером (VxD) и приложением – VxD может лишь передать приложению указатель, который может быть прямо использован приложением.

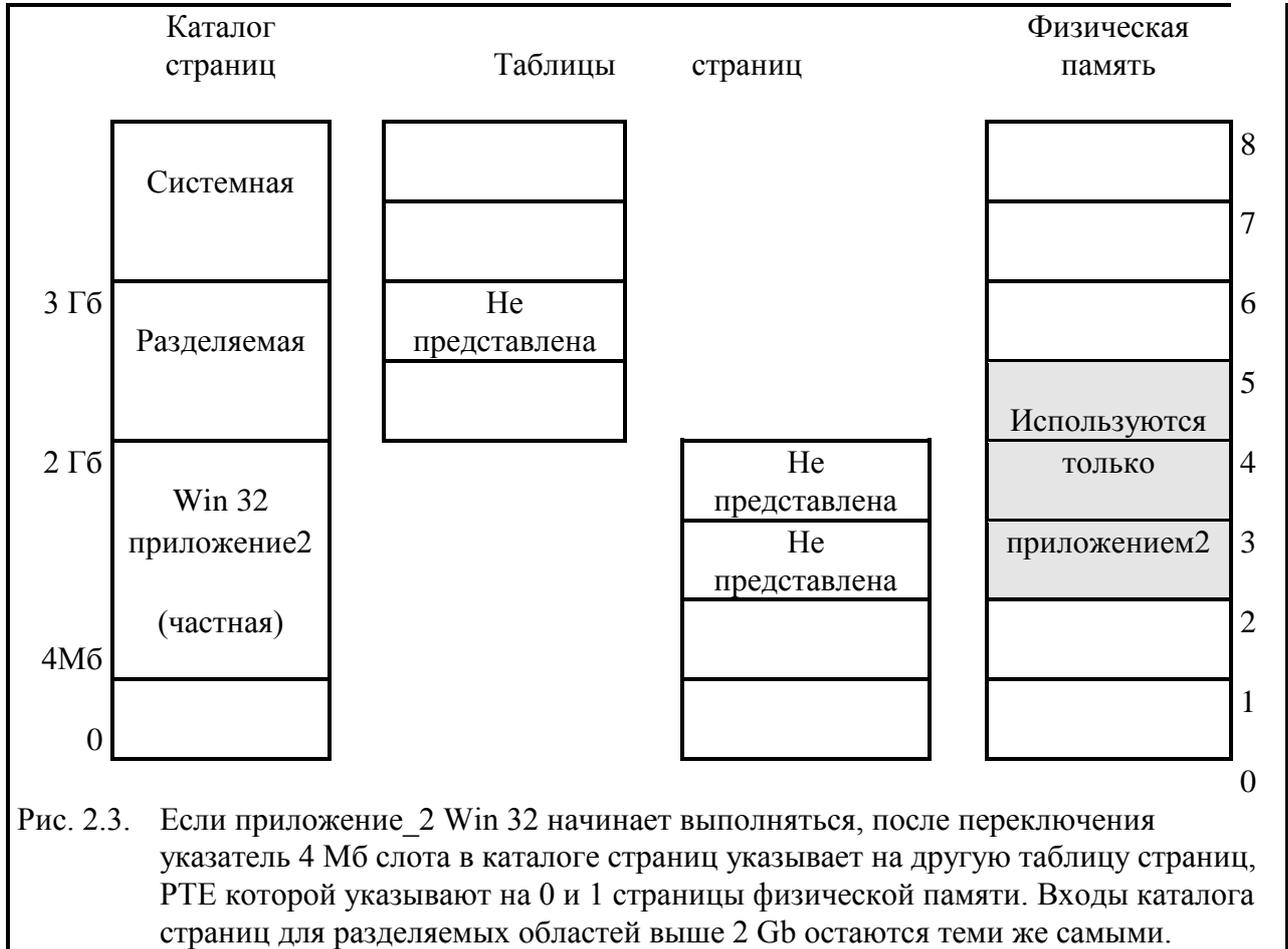
Арена DOS в линейном адресном пространстве 0-4Мб, передана приложениям DOS и небольшой части кучи (области динамической памяти – heap) Win16. Как уже говорилось, приложения DOS должны находиться здесь, так как они выполняются в режиме V86 и, следовательно, генерируют линейные адреса ниже 1Мб. Небольшая часть кучи Win16 для использования системными DLL (динамически загружаемыми библиотеками) Win16, которые занимаются выделением памяти для связи с DOS, резидентными программами, и т.д.



Windows 95 управляют входами каталога страниц (PDE) для аренды DOS таким же образом, как это делали Windows 3.x. С каждым переключением виртуальной машины, компонент V86 текущей выполняемой виртуальной машины копируется в линейное пространство, из адресов, расположенных выше 2Гб, в адреса, расположенные ниже 1Мб, простым изменением первого входа в корневом каталоге страниц.

Windows 95 более активно используют переключение каталога страниц, чем Windows 3.x. Каждый раз, когда отдельный процесс Win32 выполнен, менеджер виртуальных машин Windows 95 переключает входы каталога страниц для частной аренды, оставляя входы каталога страниц для разделяемой и системной аренды теми же. И каждый раз, когда закончила работу отдельная виртуальная машина, менеджер виртуальных машин

Windows 95 VMM переключает единственный вход каталога страниц для первых 4Мб.



## 2.6. Резюме

В этой главе было объяснено, как менеджер виртуальных машин (VMM) создает соответствующее виртуальное окружение для Win32, Win16, и приложений DOS. Менеджер виртуальных машин использует некоторые возможности процессора, связанные с привилегиями, для виртуализации доступа к устройствам, расположенным в пространстве ввода вывода или в адресном пространстве памяти, а также управляет выполнением привилегированных инструкций. Менеджер виртуальных машин использует также возможности страничного преобразования

процессора, чтобы обеспечить каждый тип приложений необходимым ему адресным пространством.

### *2.7. Контрольные вопросы*

1. Что такое захват доступа к портам ввода-вывода?
2. Что такое уровень привилегий ввода-вывода?
3. Что такое карта разрешения ввода-вывода?
4. Что такое уровень привилегий дескриптора?
5. В каком случае выполняется операция ввода-вывода приложения?
6. Каков доминирующий механизм определения привилегий в Windows 95?
7. Чем определяется уровень привилегий приложения DOS?
8. Что такое виртуализация физического устройства?
9. Кто отвечает за поддержание IOPL?
10. Что такое обработчик перехвата портов?
11. Как VMM реагирует на запрос перехваченного устройства?
12. Какие устройства PC перехватывает стандартный компонент VxD Windows 95?
13. Как перехватить обращение к устройству, отображенному на память?
14. Что такое не представленная страница?
15. Что такое обработчик отказов страниц?
16. Что такое отказ страницы?
17. Как перехватываются прерывания?
18. Как перехватываются исключения?
19. Какие инструкции перехватывает VMM?
20. Зачем перехватываются прерывания?
21. Зачем перехватываются исключения?
22. Какие инструкции перехватывает VMM в режиме V86?

23. Как виртуализуются регистры процессора?
24. Каково адресное пространство Windows 95?
25. Что такое арена?
26. Что такое частная арена?
27. Что такое разделяемая арена?
28. Что такое системная арена?
29. Что такое арена DOS?
30. Как выделяется память приложениям Win32?
31. Как выделяется память приложениям Win16?
32. Как выделяется память приложениям DOS?
33. Как передаются данные между виртуальным драйвером и приложением?

### 3. ВВЕДЕНИЕ В VxD

Хотя VxD - сокращение для *виртуального драйвера устройства*, VxD может быть чем-то большим, чем драйвер, который виртуализует конкретное устройство. Некоторые VxD действуют как драйвер устройства, но не виртуализуют его. Некоторые VxD не взаимодействуют с каким-либо устройством, они существуют для того, чтобы обеспечить обслуживание других виртуальных драйверов или приложений.

VxD может загружаться вместе с VMM (статически загружаемый) или по требованию (динамически загружаемый). Тем не менее, в обоих случаях VxD активно взаимодействует с менеджером виртуальных машин (VMM) и разделяет с ним контекст выполнения. Такие специальные отношения с операционной системой дают VxD полномочия, которые недоступны приложениям DOS и Windows. VxD имеют неограниченный доступ ко всем устройствам реальной ЭВМ, может свободно проверять структуры данных операционной системы (такие, как таблицы дескрипторов и страниц), а также может получить доступ к любой области памяти. VxD может также перехватывать программные прерывания, обращения к портам ввода-вывода и областям памяти и даже аппаратные прерывания.

Хотя приложения Windows или DOS способны выполнять некоторые задачи "низкого уровня" (например, перехватывать программные прерывания), возможности приложения всегда ограничены. Например, приложение Windows может перехватить программное прерывание, вызванное другим приложением Windows, но не прерывания, вызванные приложением DOS. VxD видит все прерывания, независимо от источника.

Для поддержания такого уровня интеграции с ядром VMM и статически и динамически загружаемые VxD должны

- соответствовать стандартной структуре,

- регистрировать свои функции в VMM, и
- обслуживать, по крайней мере, часть специального протокола сообщений.

Эта глава объясняет, как VxD загружаются и как каждый тип VxD соответствует этим фундаментальным требованиям. Далее будет показано, как VxD может использоваться, чтобы обеспечить совместимость с различными устройствами.

### *3.1. Загрузка VxD*

Windows 95 поддерживает оба типа VxD – статически и динамически загружаемые. Статически загружаемые VxD загружаются во время инициализации Windows и остаются загруженными все время работы Windows. Если VxD используется только конкретным приложением или существует только для того, чтобы обеспечить сервис некоторым приложениям, занятая им память тратится впустую, когда VxD фактически не работает. Статическая загрузка особенно раздражает разработчиков VxD, которые должны выходить из Windows и повторно их загружать для того, чтобы проверить внесенные в VxD изменения.

Windows 95 поддерживает два метода статической загрузки. Первый, поддерживаемый также Windows 3.x, заключается во внесении строки `device=VxD` в файле `System.ini`. Второй – новый для Windows 95, заключается в добавлении в регистр статически поименованного значения VxD (например, `Static VxD = pathname – путь`), в подключе `\HKLM\System\CurrentControlSet\Services\VxD`.

Динамически загружаемые VxD не загружаются автоматически при инициализации Windows, они загружаются и выгружаются под управлением приложения или другого VxD. Например, Plug and Play VxD должны быть динамически загружаемыми, потому что Windows 95

поддерживает удаление и переконфигурацию аппаратных средств ЭВМ в реальном времени. Виртуальные драйверы (VxD), которые поддерживают такие аппаратные средства, должны иметь возможность загружаться и выгружаться по мере необходимости.

Динамически загружаемые VxD удобно использовать в качестве драйверов устройств, используемых лишь конкретным приложением. Когда приложение хочет использовать устройство, оно загружает VxD, по окончании работы с устройством – выгружает VxD.

Статически и динамически загружаемые виртуальные драйверы VxD отвечают на различные наборы сообщений менеджера виртуальных машин VMM. Некоторые сообщения видны только статическим VxD, некоторые – только динамическим VxD, однако большая часть видна и тем и другим. На практике легко написать VxD, который поддерживает оба метода загрузки, просто обеспечив ответ на оба набора сообщений.

### *3.2. Базовая структура VxD*

Хотя VxD используют плоскую 32-разрядную модель памяти, коды и данные VxD все еще организованы в сегменты. (Фактически, модель адресации – база плюс смещение – является необходимым архитектурным компонентом механизма эффективной загрузки и выполнения загруженных модулей.) VxD используют следующие типы сегментов:

- инициализации реального режима,
- инициализации защищенного режима,
- со страничной организацией,
- заблокированные (без страничной организации),
- статические, и
- только отладочные.

Для каждого из этих типов сегментов, имеется сегмент кода и сегмент данных, так что VxD может иметь в общем случае 12 сегментов. Сегменты кода и данных реального режима 16-разрядные (сегментированная модель), все другие сегменты 32-разрядные (плоская модель).

Сегмент инициализации реального режима содержит код, который выполняется в начале последовательности инициализации Windows перед переключением VMM в защищенный режим. Эта ранняя фаза инициализации дает каждому статически загружаемому VxD возможность исследовать пред-Windows окружение реального режима, и затем решить, следует ли продолжить загрузку VxD. Возвращаясь с кодом возврата в AX, VxD может сообщить VMM, следует ли продолжить загрузку секции защищенного режима VxD, прервать загрузку этого VxD, или даже прервать загрузку Windows.

Большинство VxD не нуждается в процедуре инициализации реального режима, кроме PAGEFILE VxD являющегося частью менеджера виртуальных машин VMM. PAGEFILE использует некоторые вызовы DOS (int 21h), чтобы выяснить, загружен ли драйвер устройства DOS SMARTDRV. Если нет, PAGEFILE возвращается из процедуры инициализации реального режима с установленным флагом переноса, так что VMM никогда не вызовет код защищенного режима PAGEFILE.

После выполнения секции реального режима каждого статически загружаемого VxD VMM переключается в защищенный режим и дает возможность каждому статически загруженному VxD возможность выполнить код в его сегменте инициализации защищенного режима. Код инициализации защищенного режима также может возвращать код ошибки, чтобы сообщить VMM, что VxD потерпел неудачу при

инициализации. Если VxD сообщает о неудаче инициализации, VMM отмечает этот VxD как неактивный, и никогда больше не вызывает его.

Оба сегмента инициализации и реального и защищенного режима ликвидируются после завершения инициализации. Эти сегменты загружаются перед инициализацией первого VxD и не ликвидируются, пока не завершат инициализацию все VxD.

Большинство VxD размещаются в одном из сегментов. В статически загружаемом VxD, эти сегменты существуют до тех пор, пока Windows не завершают работу. В динамически загружаемом VxD они существуют до тех пор, пока VxD не будет выгружен. Как видно из названия, сегмент со страничной организацией может быть выгружен на диск менеджером виртуальных машин, в то время как заблокированный сегмент никогда не будет выгружаться. Большая часть сегментов кода и данных VxD должны иметь страничную организацию, чтобы разрешить менеджеру виртуальной памяти подкачку страниц VxD и освобождать физическую память. Только следующие элементы могут и должны выполняться в заблокированных сегментах:

- процедура управления устройством (главная точка входа VxD),
- обработчики аппаратных прерываний и все данные, к которым они обращаются,
- функции, которые могут быть вызваны обработчиком аппаратных прерываний другого VxD, (их называют асинхронными функциями).

Статические сегменты используются только динамически загружаемыми VxD, которые будут обсуждаться ниже. Статические сегменты кода и данных динамически загружаемых VxD не будут выгружены (останутся в памяти), в то время, как остальные VxD динамически выгружаются.

VMM загружает отладочные сегменты только в том случае, когда система выполняется под управлением отладчика, например WDEB386 или SoftIce/Windows. Помещая отладочный код в сегмент отладки, разработчики могут получить тот же самый выполняемый код, включая код отладки, без увеличения размера исполняемого кода. VMM загрузит отладочный код только при наличии системного отладчика, но опустит его в цикле нормальной загрузки (то есть, когда системный отладчик отсутствует).

### 3.3. Блок дескриптора устройства

Блок дескриптора устройства, или DDB (*Device Descriptor Block*), служит для связи драйвера виртуальных машин VMM с VxD. DDB содержит информацию, которая идентифицирует VxD и указатель на главную точку входа в VxD. DDB может содержать также указатели на другие точки входа, используемые приложениями или другими VxD. В табл. 3.1 показаны поля структуры DDB, которые инициализируются в VxD. Менеджер виртуальных машин VMM находит DDB VxD, и, следовательно, главную точку входа, как только он загружает VxD, ища первый экспортируемое символическое имя в модуле.

Даже когда VxD написан на Си, он не имеет главной процедуры. Вместо этого поле процедуры управления устройством в DDB содержит адрес главной точки входа в VxD. После инициализации в реальном режиме все запросы от VMM приходят в VxD через эту точку входа. VMM использует эту точку входа, чтобы уведомить VxD относительно изменения состояния виртуальной машины VM и самих Windows, и VxD выполняют свою работу, реагируя на эти события. (Эти события будут подробно обсуждаться позже.)

Таблица 3.1.

## Поля структуры блока дескриптора устройства DDB

Поле	Описание
Имя	8-байтовое имя VxD
Версия	версия VxD, не относится к версии Windows
Подверсия	версия VxD, не относится к версии Windows
Процедура управления устройством	адрес*) процедуры управления устройством
Идентификатор устройства	тот же идентификатор, который имеется в VxD или уникальное значение, назначенное фирмой Microsoft
Порядок инициализации	обычно <i>Undefined_Init_Order</i> ; чтобы вынудить произвести инициализацию перед/после конкретного VxD, следует назначить <i>Init_Order</i> в менеджере виртуальных машин VMM и добавить/вычесть 1
Таблица функций	адрес*) таблицы функций
Процедура API режима V86	адрес*) процедуры API режима V86
Процедура API защищенного режима	адрес*) процедуры API защищенного режима
	*) 32-битный адрес

Поле идентификации устройства (*Device ID*) DDB используется менеджером виртуальных машин VMM для того, чтобы идентифицировать VxD. В частности, VMM полагается на уникальный идентификатор, чтобы определить экспортируемые точки входа защищенного режима и режима V86 для API. Имеются следующие правила для подбора идентификатора устройства:

- Если ваш VxD является прямой заменой, используют идентификатор существующего VxD из заголовка файла VMM.

- Если ваш VxD не является прямой заменой и экспортирует какую-либо точку входа приложения DOS или Win16 или другого VxD, Вы должны обратиться к фирме Microsoft за уникальным идентификатором.
- Если ваш VxD не заменяет стандартный VxD и не экспортирует какую-либо точку входа приложения DOS или Win16, Вы можете использовать константу ***UNDEFINED\_DEVICE\_ID*** определенную в заголовке файла VMM.

Если VxD обеспечивает API для Win16 или приложений DOS, его DDB содержит адрес API точки входа. DDB содержит одно поле для каждого типа API: поле защищенного режима API, являющееся точкой входа 16-разрядного защищенного режима, используемой приложениями Win16, и поле режима V86 API, являющееся точкой входа, используемой приложениями DOS. Поскольку имеется только одна точка входа API для каждого из этих типов приложений, VxD обычно используют код функции в регистре для определения вызываемой функции (наподобие программного прерывания под DOS).

VxD может также экспортировать точку входа для использования другими VxD. В документации VxD это обычно называют "сервисом", а не API. Сервис отличается от API, в котором DDB содержит поле для таблицы функций, а не единственную сервисную точку входа. Таблица функций обычно содержит перечень кодов адресов функций.

Другое поле в DDB иногда используется VxD, хотя VxD и не инициализирует это поле. Поле ссылок (***Reference\_Data***) позволяет секции инициализации реального режима VxD связываться с другой секцией (защищенного режима) VxD. Когда происходит возврат из кода инициализации реального режима, VMM копирует значение из EDX в поле ***Reference\_Data*** блока данных устройства (DDB) VxD. Если код реального

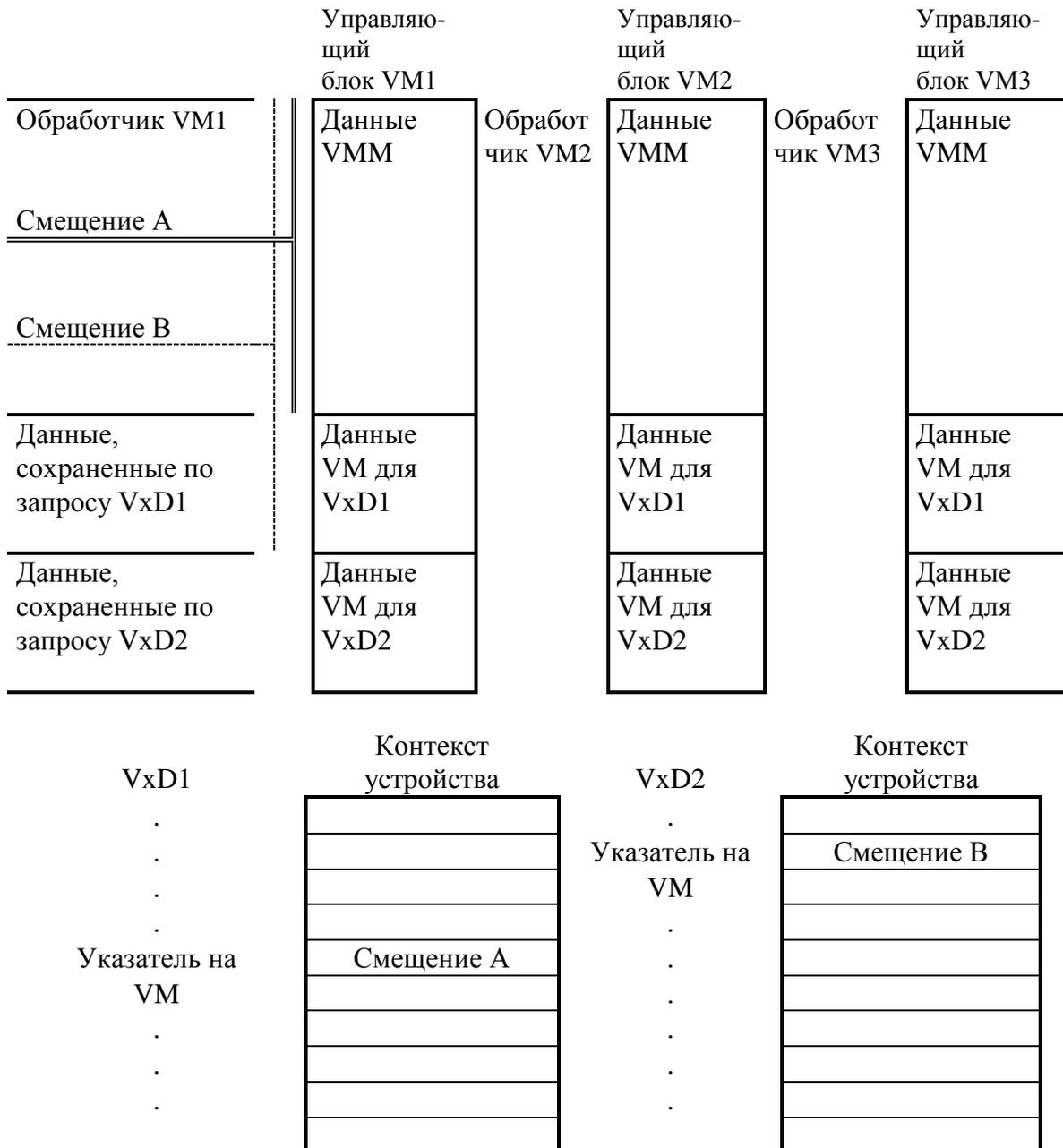
режима должен передать больше четырех байтов, он должен отвести блок памяти при помощи *LDSRV\_Copy\_Extended\_Memory* и вернуть адрес блока в EDX. Секция защищенного режима VxD может тогда использовать *Reference\_Data* как указатель на отведенный блок.

### 3.4. Поддержка структур данных

DDB – единственная структура данных, фактически требуемая VMM от VxD. Однако VxD обычно обслуживают больше, чем одно физическое устройство (например мультиплексируемые последовательные порты) и взаимодействуют более, чем с одной виртуальной машиной. Большинство VxD должны создавать свои собственные механизмы поддержки структур данных, чтобы хранить информацию о конфигурации и состоянии каждого устройства и каждой виртуальной машины.

VxD обычно используют одну или большее количество структур контекста устройств для хранения специфической для устройства информации, такой, как базовый адрес ввода вывода, запросы прерывания IRQ, и т.д. Эти структуры контекста устройства могут быть размещены статически в сегменте данных VxD (заблокированном, если используются обработчиком прерывания) или динамически посредством сервиса VMM.

Вообще, если количество устройств заведомо фиксировано, размещайте структуры устройства статически, если же количество может изменяться, размещайте их динамически. Например, все PC имеют два контроллера прямого доступа к памяти (DMA), так что виртуальный DMA драйвер объявляет статические структуры устройства в своем сегменте данных, однако, количество последовательных портов PC переменное, так что драйвер последовательного порта динамически размещает структуру каждого устройства, как только последовательный порт будет обнаружен.



**Рис. 3.1** Иллюстрирует, как блок управляющих данных (CBD) может использоваться для сохранения информации виртуальной машины для каждого из нескольких устройств

Если Вы динамически размещаете вашу структуру устройства во время выполнения, используйте сервисную функцию VMM *HeapAllocate*, которая подобна функции *malloc* языка Си. Однако, если ваша структура устройства включает большой буфер (4Кб или больший), следует включить непосредственно в структуру устройства только указатель на

буфер, а затем отвести большой буфер, используя сервисную функцию *\_PageAllocate*. Как правило, используют функцию *\_HeapAllocate* для отведения небольших областей и *\_PageAllocate* для отведения больших областей. Границей между большой и маленькой областью является значение 4Кб.

В то время, как служебная информация о каждом устройстве хорошо знакома разработчикам драйверов устройств, служебная информация о каждой виртуальной машине, или о каждой паре устройство/виртуальная машина менее знакома разработчикам. К счастью VxD может попросить, чтобы менеджер виртуальных машин (VMM) хранил данные виртуальной машины от имени VxD. Менеджер виртуальных машин (VMM) сам отводит и использует управляющий блок для каждой виртуальной машины. VxD может использовать сервис VMM резервирования его собственной области данных о виртуальной машине в пределах управляющего блока виртуальной машины VM.

Для резервирования этого места в управляющем блоке, VxD запрашивает сервисную функцию VMM *\_Allocate\_Device\_CB\_Area* во время инициализации, запрашивая блок соответствующего размера. Менеджер виртуальных машин (VMM) возвращает смещение отведенного блока в пределах всего управляющего блока. Как только VxD запросил эту область, менеджер виртуальных машин (VMM) будет резервировать ее по одному и тому же смещению в управляющем блоке каждой виртуальной машины (VM). Поскольку VxD всегда будет иметь доступ к обработчику текущей виртуальной машины, а обработчик виртуальной машины (VM) фактически является стартовым адресом управляющего блока VM, VxD будет иметь доступ к данным этого управляющего блока. На рис. 3.1 показано, как данные управляющего блока (CBD) могут использоваться для сохранения информации о состоянии виртуальной машины.

Кроме данных о виртуальной машине, некоторым VxD необходимы также данные о потоках. Причина заключается в том, что Windows 95 управляет потоками, а не виртуальными машинами, а системная виртуальная машина может иметь более одного потока. Механизм хранения данных о потоке похож на тот, который используется для хранения данных о виртуальной машине. VxD отводит место для хранения данных о потоке в процессе своей инициализации, вызывая сервисную функцию *\_AllocateThreadDataSlot*. Эта функция возвращает смещение слота данных потока относительно структуры данных, называемой управляющим блоком потока или *Thread Control Block (THCB)*. Менеджер виртуальных машин (VMM) обеспечивает THCB текущего выполняемого потока, когда вызывается управляющая процедура устройства VxD с сообщениями, связанными с потоком. VxD может получить также управляющий блок текущего выполняемого потока THCB путем вызова сервисной функции VMM *Get\_Cur\_Thread\_Handle*.

В отличие от *\_Allocate\_Device\_CB\_Area*, которая может резервировать области данных различного размера, *\_AllocateThreadDataSlot* всегда отводит 4 байта для сохранения данных о потоке. Если потоковые данные вашего VxD не помещаются в 4 байтах, используйте эти 4 байта для хранения указателя на структуру большего размера. Ваш VxD должен отводить структуру большего размера, когда поток уже создан (рис. 3.2).

Чтобы просматривать или изменять состояние виртуальной машины, VxD просматривает или изменяет поля в другой важной структуре данных – структуре регистра клиента. Эта структура содержит текущие регистры и флаги виртуальной машины.

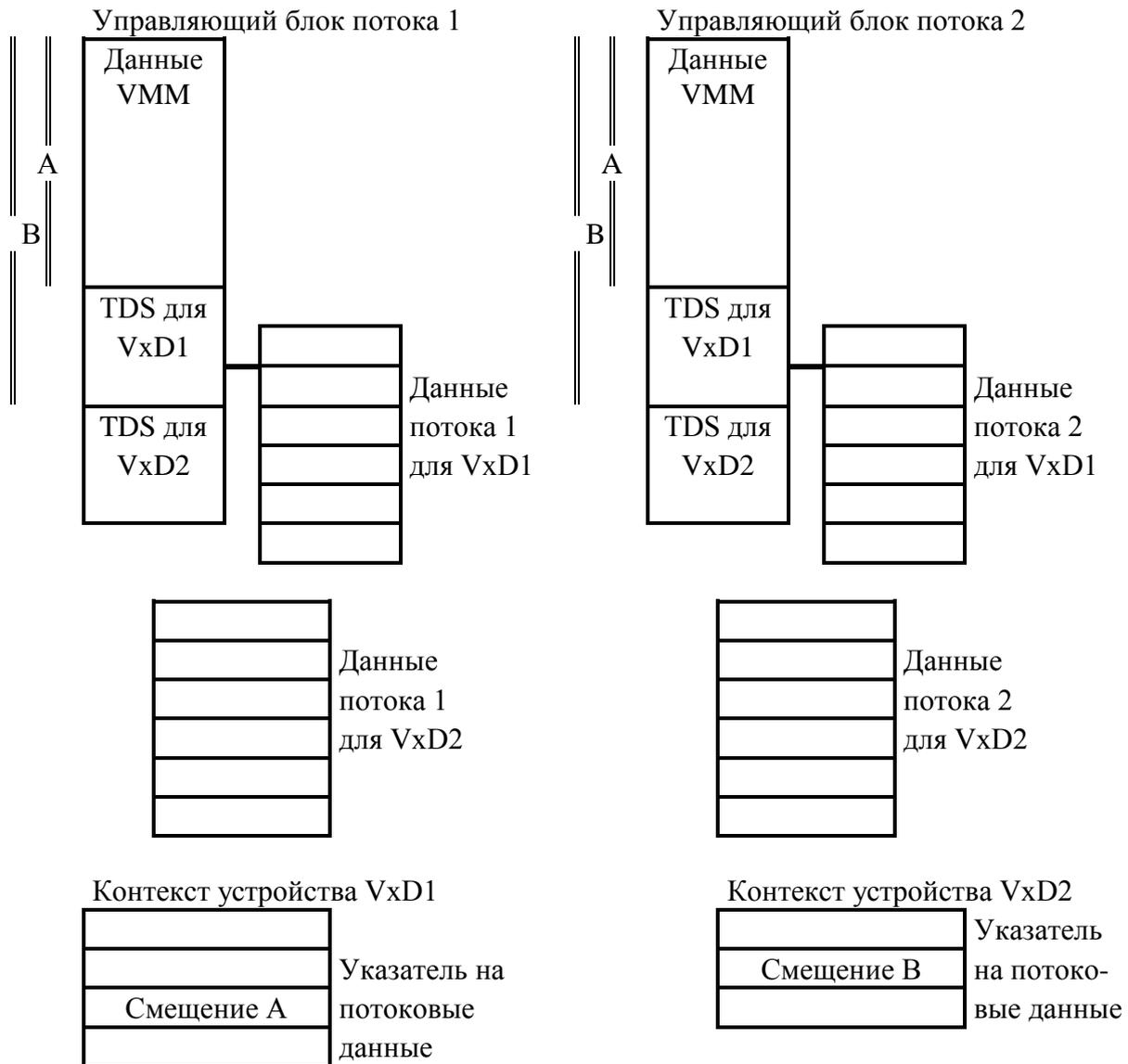


Рис. 3.2. Показывает, как слот данных потока (TSD) может быть использован для хранения потоковых данных

Обычно VxD интересуется состоянием виртуальной машины, если он обеспечивает API приложений защищенного режима или режима V86. Такой VxD получает его вход и обеспечивает его вывод при помощи этого регистра клиента. Прежде, чем VxD вызовет точку входа API, менеджер виртуальных машин VMM устанавливает в EBP указатель на структуру регистра клиента. VxD может также найти адрес структуры регистра

клиента через `CB_Client_Pointer` в управляющем блоке виртуальной машины. На рис. 3.3. показаны эти взаимосвязи.

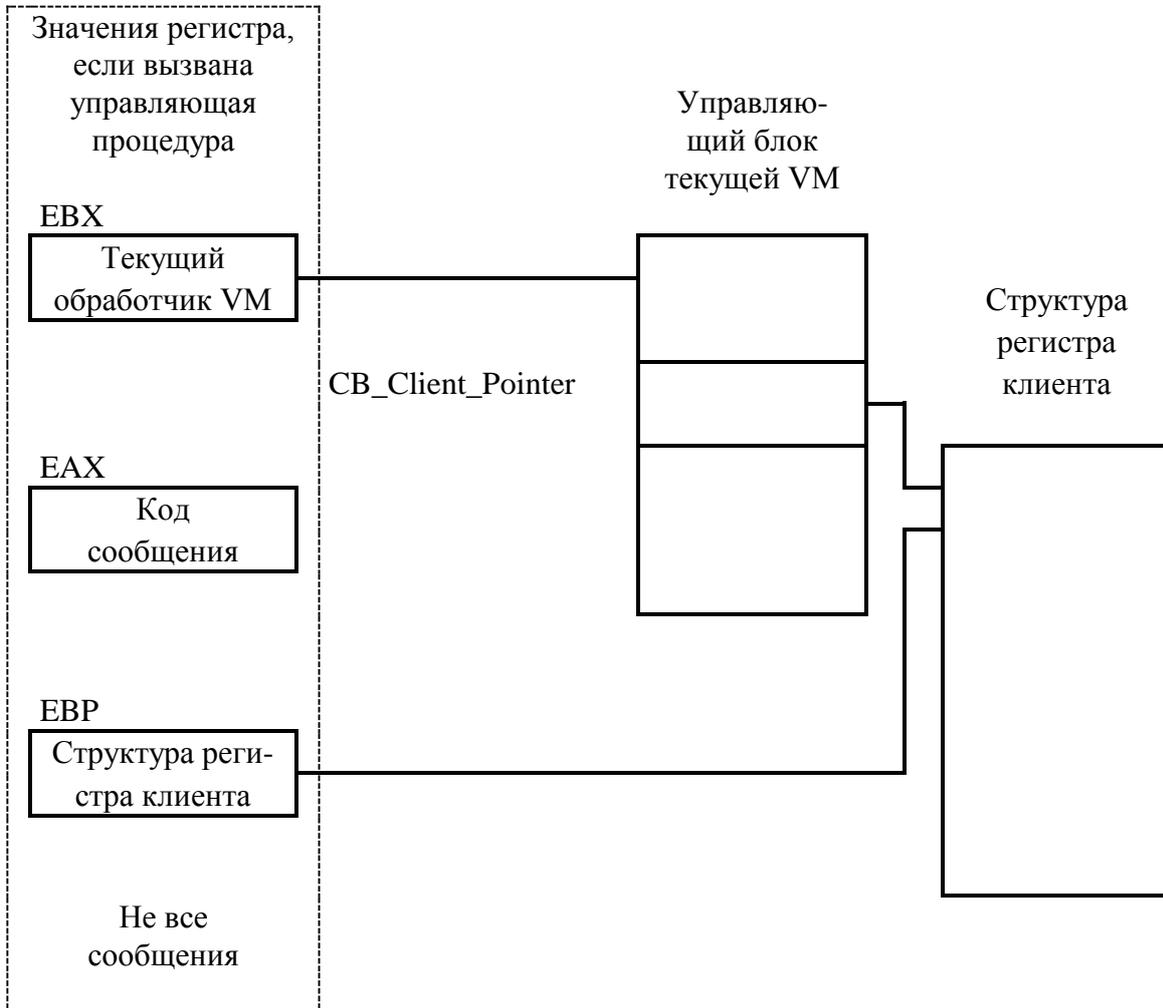


Рис. 3.3. Иллюстрирует взаимосвязь между обработчиком текущей VM, управляющим блоком VM и структурой регистра клиента

### 3.5. Уведомление о событии

Как только инициализация реального режима завершена, менеджер виртуальных машин VMM уведомляет VxD о значимых событиях через специальный интерфейс сообщений. Чтобы послать сообщение VxD, VMM получает адрес управляющей процедуры устройства VxD от блока дескриптора устройства (DDB) VxD и вызывает эту процедуру с кодом сообщения в EAX и обработчиком текущей VM в EBX. Тогда управляющая процедура осуществляет переход, соответствующий сообщению. VMM использует этот интерфейс для уведомления VxD о выполнении инициализации, завершения и изменения состояния VM.

Хотя интерфейс сообщений VxD концептуально подобен интерфейсу сообщений WinProc, их реализация совершенно различна и несовместима.

Грубо две дюжины сообщений могут быть разделены на восемь основных категорий. Сообщения и их категории показаны в табл. 3.2. Сообщения в категориях инициализации и завершения всегда посылаются в указанном в списке порядке. Более подробный список сообщений и их регистрируемых параметров и кодов возврата может быть найден в документации к DDK Windows 95.

Многие VxD обрабатывают только небольшую часть этих сообщений. Пример VxD в следующей главе иллюстрирует обработку наиболее часто обрабатываемых сообщений. Большинство этих сообщений отмечает важные события в жизни VxD или VM. В следующем разделе объясняется, какое отношение эти сообщения имеют к нормальному циклу жизни VxD и как VM их обслуживает.

Таблица 3.2.

Сообщения уведомления о событиях, передаваемые VMM в VxD

Категория сообщений	Сообщение	Описание
Инициализация системы	Sys_Critical_Init	С этого момента и далее прерывания запрещены. Минимальная обработка
	Device_Init	Системная VM уже загружена. Большую часть инициализации VxD выполняют здесь.
	Init_Complete	Обработка необходимая после того, как все VxD выполняют Device_Init.
Завершение работы системы	System_Exit	Системная VM уничтожена, но все еще в памяти.
	Sys_Critical_Exit	Системной VM больше нет в памяти. Прерывания запрещены.
Инициализация VM	Create_VM	VxD инициализирует данные VM.
	VM_Critical_Init	Прерывания запрещены.
	VM_Init	VM полностью создана. VxD может теперь вызывать код VM.
	Sys_VM_Init	Эквивалентно VM_Init, но VM системная.

Таблица 3.2. (продолжение)  
Сообщения уведомления о событиях, передаваемые VMM в VxD

Категория сообщений	Сообщение	Описание
Завершение работы VM	Query_Destroy	Ненормальное завершение работы VM. Если VM не должна быть уничтожена, возвращается установленный флаг переноса.
	VM_Terminate	Нормальное завершение работы VM. VM все еще существует, так что VxD может вызывать код VM.
	Sys_VM_Terminate	Эквивалентно VM_Terminate, но VM системная.
	VM_Not_Executable	Посылается и для нормального и для ненормального завершения. VM все еще в памяти, но не может выполняться.
	Destroy_VM	VM больше нет в памяти.
Изменение состояния VM	VM_Suspend	VM временно приостанавливается другим VxD. VxD должен освободить ресурсы, связанные с этой VM.
	VM_Resume	VM возобновляет работу после приостановки.
	Set_Device_Focus	VM имеет фокус клавиатуры/мыши.
	Begin_PM_App	VM запустила приложение защищенного режима.
	End_PM_App	VM завершила приложение защищенного режима.

Таблица 3.2 (окончание)

Сообщения уведомления о событиях, передаваемые VMM в VxD

Категория сообщений	Сообщение	Описание
Инициализация потока	Create_Thread	Создается новый поток. Отведение и инициализация ТНСВ данных.
	Thread_Init	Новый поток создан и в настоящее время выполняется.
Завершение потока	Terminate_Thread	Поток собирается завершиться. Освобождение потоковых ресурсов.
	Thread_Not_Executable	Поток завершается и не будет больше выполняться.
	Destroy_Thread	Поток уничтожен.
Разное	Reboot_Processor	Обрабатывается только виртуальным драйвером клавиатуры.
	Debug_Query	Производится от имени отладчика. VxDs показывает состояние.

### 3.6. Сообщения инициализации и завершения статически загружаемого VxD

Статически загружаемый VxD загружается при инициализации Windows и выгружается, когда Windows завершают свою работу. Во время инициализации Windows статически загружаемый VxD получает три сообщения, отмечающих каждую стадию инициализации Windows. В ответ на любое из этих трех сообщений, VxD может сообщить о неудаче, возвращая установленный флаг переноса. После этого Windows выгрузят VxD, и VxD не получит больше никаких сообщений.

Первая стадия инициализации Windows отмечена в сообщении *Sys\_Critical\_Init*. В это время прерывания запрещены, так что, если ваше устройство требует непрерываемой инициализации, выполняйте ее здесь. Если VxD экспортирует услуги другим VxD, он должен выполнить всю инициализацию, необходимую для выполнения этих услуг в обработчике *Sys\_Critical\_Init*, потому что другие VxD могут запросить экспортируемые

услуги сразу после того, как экспортирующий VxD обработает это сообщение. Если VxD виртуализует адаптер, размещенный в памяти, используемый приложениями DOS, то он должен здесь зарезервировать страницы в адресном пространстве V86. (Например, виртуальный адаптер дисплея резервирует страницы для видеобуфера в адресном пространстве каждой виртуальной машины, обычно по адресам A0000h-C0000h.)

Все VxD должны отложить любые другие действия до следующей стадии. Обратите внимание на то, что функции типа *Simulate\_Int* или *Exec\_Int*, которые выполняют код в VM, недоступны в это время, потому что никакие виртуальные машины еще созданы. (Объяснение роли функций *Simulate\_Int* и *Exec\_Int* будет дано в ниже.)

Следующее сообщение, *Device\_Init*, уведомляет VxD о фазе инициализации, которая выполняется после того, как менеджер виртуальных машин VMM создал системную VM. Большая часть установки VxD выполняется в этой фазе. В это время VxD должен разместить контекст устройства и блока управления памятью, перехватить порты ввода-вывода и прерывания.

*Init\_Complete* отмечает последнюю фазу инициализации системы. Обычно только VxD, которые отводят страницы в адресном пространстве V86, должны отвечать на это сообщение.

Завершение работы Windows также происходит в три фазы. Когда система нормально завершает работу (то есть, не наступает крах системы), сначала завершает работу системная VM, выдавая сообщение *Sys\_VM\_Terminate*. Так как системная VM не была разрушена, сервисные функции *Simulate\_Int* и *Exec\_Int* все еще доступны, если VxD надо выполнять код в системной VM. Следующее сообщение в последовательности закрытия – *System\_Exit*, которое выдается и при нормальном и при ненормальном завершении. В это время прерывания

разрешены, однако системная VM уже разрушена, так что *Simulate\_Int* и *Exec\_Int* больше недоступны. Большинство VxD закрываются во время *System\_Exit*, закрывая свои устройства. Последнее сообщение – *Sys\_Critical\_Exit* посылается с запрещенными прерываниями. Большинство VxD не обрабатывают это сообщение.

### 3.7. Сообщения инициализации и завершения динамически загружаемых VxD

Динамически загружаемый VxD не видит сообщений инициализации системы (*Sys\_Critical\_Init*, *Device\_Init*, и *Init\_Complete*) потому что он не был загружен, когда эти сообщения выдавались. Однако, VMM обеспечивает аналогичное сообщение для динамического VxD в процессе процедуры его загрузки – *Sys\_Dynamic\_Device\_Init*, и другого сообщения, когда VxD выгружается – *Sys\_Dynamic\_Device\_Exit*.

Динамический VxD обрабатывает сообщение *Sys\_Dynamic\_Device\_Init* так же, как статический VxD обработал бы сообщения инициализации системы – выполняя инициализацию основного устройства, перехватывая порты ввода-вывода, устанавливая обработчики аппаратных прерываний и т.д. Следует обратить внимание на то, что некоторые сервисные функции VMM доступны только в процессе инициализации системы и поэтому не могут использоваться динамическими VxD (см. перечень сервисных функций DDK Windows 95). Динамический VxD может сообщить о неудачной попытке загрузки, возвращаясь из обработчика сообщения *Sys\_Dynamic\_Device\_Init* с установленным флагом переноса.

Хотя статически загружаемые VxD получают несколько сообщений завершения системы, статические VxD часто небрежны в отношении освобождения ресурсов в процессе завершения, так как сами Windows

завершают свою работу. Напротив, динамический VxD должен очень аккуратно освобождать любые ресурсы, которые он занимал. Это касается освобождения портов ввода-вывода, выгрузки обработчиков аппаратных прерываний, и освобождения сервисных функций. Кроме того, динамический VxD должен отменить все незавершенные тайм-ауты и события, возникшие во время *Sys\_Dynamic\_Device\_Exit*, иначе VMM закончит тем, что будет вызывать код, который уже выгружен, и система, вероятно, потерпит крах.

Статические сегменты кода и данных могут использоваться для решения некоторых проблем, с которыми может сталкиваться динамический VxD при освобождении ресурсов. Например, иногда VMM не обеспечивает сервиса "освобождения" для какого-то ресурса, и освобождение может быть неудачным. В этих случаях код, использующий этот ресурс, должен находиться в статическом кодовом сегменте и не должен выполнять никаких действий, если остальная часть VxD не загружена. При последующей загрузке VxD должен повторно использовать уже занятый ресурс, вместо того, чтобы занимать ресурс снова.

### *3.8. Сообщения об изменении состояния VM*

Здесь имеет место другой набор сообщений о жизненном пути виртуальной машины (VM). Создание новой VM также происходит в три фазы, каждая со своим собственным сообщением: *Create\_VM*, *VM\_Critical\_Init* и *VM\_Init*. Обработчик VM для каждого из сообщений находится в EBX.

Когда VxD получает первое сообщение, *Create\_VM*, он должен инициализировать данные, связанные с VM. *VM\_Critical\_Init* отмечает следующую фазу. Реакция с ошибкой на сообщение *VM\_Critical\_Init*

(возвращается установленный флаг переноса) инициирует последовательность завершения, начинающуюся с *VM\_Not\_Executable*. (Эта последовательность не совпадает с последовательностью завершения, инициированной неудачей *VM\_Create*.) Заключительная стадия создания – *VM\_Init*. В это время VM уже создана, и в ней становятся доступными для вызова программные прерывания *Simulate\_Int* и *Exec\_Int*.

Уничтожение VM также совершается в три фазы, снова с обработчиком VM в EBX. Существующая VM изящно завершается сообщением *VM\_Terminate*, которым VM сообщает, что "собирается умирать". (Неправильное завершение сначала произведет *Query\_Destroy*, о чем будет сказано далее.) Здесь VxD должен выполнить действия, связанные с *Simulate\_Int* или *Exec\_Int*, пока виртуальная машина еще существует. Следующая фаза – *VM\_Not\_Executable* – выполняется и при нормальном и при ненормальном выходе. Регистр EDX содержит значения флагов, которые указывают на фактическую причину завершения.

Эти значения флагов приведены в табл. 3.3. Поскольку VM уже закрыта, *Simulate\_Int* и *Exec\_Int* недоступны. Последняя фаза обозначена *Destroy\_VM*. Если VxD заботит истинная причина завершения работы VM, и ему не надо использовать *Simulate\_Int* или *Exec\_Int*, он может отвечать только на это заключительное сообщение.

Прежде, чем SHELL VxD ненормально закроет VM (например, в ответ на запрос пользователя), он пошлет сообщение *Query\_Destroy*. VxD может ответить на это сообщение с установленным флагом переноса, чтобы указать, что SHELL не должен уничтожать VM. В этом случае, VxD должен также сообщить пользователю о возникновении проблемы, используя сервисные сообщения SHELL.

В дополнение к событиям запуска и закрытия VM, VxD также уведомляется относительно планирования событий, которые изменяют

выполняемую в настоящее время VM. Сообщения *VM\_Suspend* и *VM\_Resume* посылаются VxD, если планировщик VMM приостанавливает или возобновляет выполнение VM.

Хотя в документации DDK говорится о том, что следует освобождать любые ресурсы, связанные с временно приостановленными VM по получении *VM\_Suspend*, только некоторые из VxD, созданные на базе DDK, отвечают на сообщения *VM\_Suspend* и *VM\_Destroy*. Виртуальный драйвер дисплея (VDD) отвечает на *VM\_Suspend*, разблокируя страницы видеопамати, и на *VM\_Resume*, снова блокируя эти страницы. Виртуальный COM-драйвер (VCD) отвечает на *VM\_Suspend*, сбрасывая ожидающие прерывания последовательного порта, принадлежащего приостановленной VM.

Таблица 3.3.

Значения флагов, содержащиеся в регистре EDX, указывающие причину завершения

Флаг	Описание
VNE_Crashed	VM потерпела крах.
VNE_Nuked	VM уничтожена, но все еще активна.
VNE_CreateFail	VxD потерпел неудачу в <i>Create_VM</i>
VNE_CrInitFail	VxD потерпел неудачу в <i>VM_Critical_Init</i>
VNE_InitFail	VxD потерпел неудачу в <i>VM_Init</i>
VNE_Closed	VM закрытая должным образом уничтожена.

### 3.9. Поточковые сообщения

Это набор сообщений о жизненном цикле потоков – единиц управления задачами, используемых планировщиком VMM Windows 95. Это сообщения: *Create\_Thread*, *Thread\_Init*, *Terminate\_Thread*, *Thread\_Not\_Executable* и *Destroy\_Thread*. Однако, эти сообщения не посылаются начальным потоком VM, а только потоками, созданными впоследствии VM. Как уже говорилось, виртуальные машины DOS имеют

точно один поток каждая, поэтому, хотя создание VM DOS завершается созданием нового потока, VMM не посылает сообщения *Create\_Thread* (но посылает при этом сообщение *Create\_VM*.)

Так же, как и виртуальные машины, потоки создаются и разрушаются поэтапно. Первое сообщение – *Create\_Thread*, посылается в начале процесса создания потока. EDI содержит обработчик (THCB) создаваемого потока (который не является выполняемым в настоящее время потоком). VxD может возвращать установленный флаг переноса, и VMM не будет создавать поток. Обычно здесь VxD размещает и инициализирует потоко-ориентированные данные. Этап внешнего отведения памяти необходим, если 4-х байтов данных на поток в THCB (отведенных в процессе инициализации VxD) недостаточно. В этом случае, структура данных потока размещается на этапе *Create\_Thread*, и данные потока в THCB используются для того, чтобы хранить указатель на эту вновь отведенную структуру.

Как только поток полностью создан, VMM посылает сообщение *Thread\_Init*. EDI опять содержит обработчик вновь созданного потока, но теперь новый поток также является потоком, выполняемым в настоящее время. VxD должен задержать любую инициализацию, которая требует выполнения нового потока, пока он не получит этого сообщения.

Разрушение потока также влечет за собой многократные сообщения: *Terminate\_Thread*, *Thread\_Not\_Executable*, и *Destroy\_Thread*. Когда первое сообщение, *Terminate\_Thread*, послано, поток "собирается быть завершенным", но все еще способен выполняться. Обычно VxD отвечает на это сообщение, освобождая любые ресурсы, связанные с потоком. Следующее сообщение, *Thread\_Not\_Executable*, посылается, когда поток больше не может выполняться. Последнее сообщение *Destroy\_Thread*

выдается после того, как поток фактически был разрушен и дает VxD последний шанс для освобождения потоко-ориентированных ресурсов.

### 3.10. Отличие от Windows 3.x

Windows 3.x использовали только три типа сегментов: инициализированные в реальном режиме, инициализированные в защищенном режиме, и заблокированные (без страничной организации). VMM Windows 3.x никогда не занимались подкачкой кода или данных виртуальных драйверов (VxD).

Windows 3.x не поддерживают динамическую загрузку VxD – используется только статическая загрузка. Статическая загрузка определена через `device =` заявление в разделе `[386Enh]` файла `System.ini`, так же, как и в Windows 95.

Windows 3.x не поддерживают потоки. Это означает, что нет никакой надобности в потоковых данных, нет *Allocate\_Thread\_Data\_Slot*, и нет никаких потоко-ориентированных сообщений.

### 3.11. Резюме

Несмотря на сотни функций, поддерживаемых VMM и другими VxD, для многих VxD не надо знать больше того, что описано в этой главе. Если Вы не делаете что-то очень специальное (например, не пишете замену для VMM), вы, вероятно, никогда не будете нуждаться больше чем в дюжине функций этого API.

### 3.12. Контрольные вопросы

1. Что такое статически загружаемый VxD?
2. Что такое динамически загружаемый VxD?
3. Какие дополнительные полномочия имеет VxD по сравнению с приложениями?
4. Может ли VxD перехватывать программные прерывания?
5. Может ли VxD перехватывать обращения к портам ввода-вывода?
6. Может ли VxD перехватывать обращения к памяти?
7. Может ли VxD перехватывать аппаратные прерывания?
8. Каковы преимущества статической загрузки VxD?
9. Каковы преимущества динамической загрузки VxD?
10. Каковы два способа статической загрузки VxD?
11. Какие типы сегментов используют VxD?
12. Что такое сегмент инициализации реального режима?
13. Что такое сегмент инициализации защищенного режима?
14. Что такое сегмент со страничной организацией?
15. Что такое заблокированный сегмент?
16. Что такое статический сегмент?
17. Что такое сегмент отладки?
18. Зачем нужен код сегмента инициализации реального режима?
19. Как передается код возврата для VMM?
20. Для чего нужен код возврата для VMM?
21. Что происходит с сегментами инициализации после завершения инициализации?
22. Какие элементы VxD должны выполняться в заблокированных сегментах?
23. В каком случае VMM загружает отладочные сегменты?
24. Для чего нужен блок дескриптора устройства?

25. Что такое главная точка входа VxD?
26. Для чего нужно поле «порядок инициализации»?
27. Для чего нужно поле идентификации устройства?
28. Каковы правила подбора идентификатора устройства?
29. Что такое «сервис» VxD?
30. Как динамически разместить структуры устройств?
31. Какую функцию лучше использовать для отведения небольшой области памяти?
32. Какую функцию лучше использовать для отведения большой области памяти?
33. Что такое «большая область памяти» при отведении?
34. Как сохранить данные и виртуальной машине для VxD?
35. Как отвести место для данных о VM в управляющем блоке VM?
36. Что такое поток?
37. Как сохранить данные о потоке?
38. Как отвести место для данных о потоке?
39. Что такое управляющий блок потока?
40. Что такое интерфейс сообщений?
41. Как VMM посылает сообщение VxD?
42. Как передается в VxD код сообщения?
43. Как передается в VxD текущий обработчик VM?
44. Категории сообщений.
45. Сообщения инициализации.
46. Сообщения завершения.
47. Сообщения инициализации статически загружаемого VxD.
48. Сообщения инициализации динамически загружаемого VxD.
49. Сообщения завершения статически загружаемого VxD.
50. Сообщения завершения динамически загружаемого VxD.

51. Сообщение *Sys\_Critical\_Init*.
52. Сообщение *Device\_Init*.
53. Сообщение *Init\_Complete*.
54. Сообщение *Sys\_VM\_Terminate*.
55. Сообщение *System\_Exit*.
56. Сообщение *Sys\_Critical\_Exit*.
57. Сообщение *Sys\_Dynamic\_Device\_Init*.
58. Сообщение *Sys\_Dynamic\_Device\_Exit*.
59. Сообщение *Create\_VM*.
60. Сообщение *VM\_Critical\_Init*.
61. Сообщение *VM\_Init*.
62. Сообщение *VM\_Not\_Executable*.
63. Сообщение *VM\_Terminate*.
64. Сообщение *Query\_Destroy*.
65. Сообщение *Destroy\_VM*.
66. Сообщение *VM\_Suspend*.
67. Сообщение *VM\_Resume*.
68. Сообщение *Create\_Thread*.
69. Сообщение *Thread\_Init*.
70. Сообщение *Terminate\_Thread*.
71. Сообщение *Thread\_Not\_Executable*.
72. Сообщение *Destroy\_Thread*.

## 4. СТРУКТУРА VxD

В этой главе представлен "скелет" VxD, который обеспечивает немного функциональных возможностей, но определяет основную структуру будущих VxD. Этот скелетный VxD будет лишь контролировать создание и уничтожение VM и потоков, а также будет выводить на печать информацию о VM и о потоках в ходе этих процессов. Эта выходная информация будет посылаться отладчику и в файл – эта техника может использоваться в VxD для того, чтобы сохранить информацию для отладки.

Здесь представлено два различных подхода к разработке VxD на языке Си: один использует DDK Windows 95, другой – продукт фирмы Vireo Software VToolsD. VToolsD предоставляет большой стартовый задел, автоматически производя файл сборки (makefile) и прототип файла Си. VToolsD также не требует никаких модулей на языке ассемблера. Напротив, DDK-технология требует одного файла на языке ассемблера. В этой главе больший акцент делается на DDK-технологии, так как он более сложен.

### 4.1. Инструменты для создания VxD

Во времена Windows 3.x VxD почти всегда был написан на ассемблере, просто потому, что VxD представляли собой программы с плоской 32-разрядной моделью памяти, и имелось немного доступных 32-разрядных компиляторов с языка Си. Теперь, когда 32-разрядные компиляторы являются нормой, можно писать VxD на Си. Однако, стандартного набора 32-разрядного компилятора и компоновщика недостаточно.

Необходимы также включаемые (.h) файлы для VMM и других VxD функций, а также специальная библиотека для взаимодействия с VMM и

другими VxD. Обычно в библиотеке содержится код "склеивания", который преобразует используемые VMM и другими VxD функциями вызовы, основанные на регистрах, в интерфейс вызовов Си. Включаемые файлы и VMM библиотека доступны из двух различных источников: DDK Windows 95 (*Device Driver Kit*) и пакет VToolsD.

И DDK Windows 95 и VToolsD идут с инструментами, которые необходимы для написания VxD на Си – добавляются 32-разрядный компилятор и компоновщик. VToolsD явно поддерживает компиляторы Borland и Microsoft, в то время как DDK Windows 95 поддерживает только Microsoft, хотя его можно вынудить работать с Borland. VToolsD включает некоторые другие особенности, которых нет в DDK Windows 95. Одна из них QuickVxD – мастер VxD, который быстро генерирует скелет VxD, включая исходный файл Си, заголовочный файл, и файл сборки (makefile). VToolsD также включает runtime библиотеку Си для VxD. Эта дополнительная библиотека полезна, потому что VxD уже не может пользоваться стандартной runtime библиотекой 32-разрядного компилятора Си. Предположения стандартных библиотек компилятора относительно runtime окружения не соответствуют истине для VxD.

Хотя DDK технически обеспечивает все, что необходимо для написания VxD на языке Си, VToolsD делает этот процесс существенно более легким. "Склеивающая" библиотека VMM, обеспечивающая и VToolsD и DDK, решает только половину проблемы. Она позволяет VxD, написанному на Си, вызвать VMM и другие VxD функции, использующие параметры, основанные на регистрах. Однако, только VToolsD решает проблему параметров, основанных на регистрах в другом направлении. Все сообщения, посланные управляющей процедуре устройства вашего VxD, а также обратные вызовы (перехват портов, прерывания, обработчик ошибок, и т.д.), передают параметры в регистрах. При использовании

DDK, Вы должны или писать маленькие функции на ассемблере или включать операторы ассемблера непосредственно в текст Си, чтобы извлечь эти регистровые параметры. VToolsD же обеспечивает стандартную для Си передачу параметров через стек и позволяет писать обработчики сообщений и обратные вызовы (callback) функций на Си.

Даже не используя DDK инструменты разработки, можно найти там много ценного. DDK содержит исходный код почти дюжины VxD, которые поставляются с Windows 95 – от виртуального драйвера дисплея до виртуального драйвера прямого доступа к памяти (DMA) и виртуального драйвера NetBios. Если Вы планируете писать VxD, для поддержания новых аппаратных средств, подобных существующему устройству, целесообразно использовать DDK и подгонять существующие VxD под ваши устройства. Даже если Вы создаете новый фирменный VxD, изучение существующих VxD весьма полезно, а DDK похоже единственный источник для изучения нетривиального реального мира VxD.

Для работы необходим также отладчик, для отладки ваших VxD, так как отладчик прикладного уровня, поставляемый со стандартным компилятором не в состоянии этого обеспечить. Только два пакета в состоянии отлаживать VxD: отладчик WDEB386, включенный в DDK и SoftIce/Windows фирмы NuMega Technologies. Использовать ли WDEB386 или Softice – в значительной степени вопрос вкуса, денег, и предпочтений разработчика. Хотя оба достаточно мощны для отладки VxD, Softice более дружелюбный: WDEB386 требует терминала, Softice - нет, кроме того, SoftIce/Windows позволяет производить отладку на уровне исходного языка Си, WDEB386 показывает Вам только ассемблер.

## 4.2. Исходные файлы версии "DDK "

"DDK" версия скелетного VxD состоит из двух исходных файлов:

- SKELCTRL.ASM, который содержит *блок дескриптора устройства* (DDB) и *управляющую процедуру устройства*, имеющиеся в каждом VxD;
- SKELETON.C, который содержит функции обработчика сообщений, вызываемые *управляющей процедурой устройства*.

Кроме того, фирма Vireo (изготовитель VToolsD) предоставляет файл VXDCALL.C, который содержит заплату, устраняющую ошибку в компиляторе Microsoft VC ++ 4.1.

Хотя не обязательно размещать DDB и управляющую процедуру устройства в файле на языке ассемблера (VToolsD этого не делает), лучше это делать так. Они очень малы и легко могут быть написаны на ассемблере, а помещение их в файл Си потребовало бы написания сложного макроса и включения ассемблерного фрагмента.

Как уже говорилось, когда модуль Си вызывает сервисную функцию VMM или VxD, функция на языке ассемблера должна брать параметры из стека и размещать их в соответствующих регистрах, как этого ожидает соответствующая сервисная функция. Библиотека VXDWRAPS.CLB в DDK обеспечивает оболочки для некоторых часто используемых сервисных функций VMM и VxD, но SKELETON.VXD использует некоторые функции, которые не содержатся в этой библиотеке.

В этой главе сосредоточено внимание на том, как SKELETON.C (листинг 4.1) использует функции в библиотеке оболочки, а не функции оболочки непосредственно.

Если вы используете Microsoft VC++ 4.1 для разработки собственного VxD, вам надо будет прикомпоновать в ваш VxD еще один файл – VXDCALL.C. Без этого модуля ошибка в компиляторе версии 4.1

сделает созданный VxD бесполезным. Короче говоря, компилятор генерирует неправильный код, когда enums используются во вложенных операторах ассемблера: макрокоманда VMMSCall в VMM.H использует enums. VxD, сгенерированный с этим неправильным кодом вызывает появление runtime сообщения об ошибке, "Неподдерживаемая сервисная функция xx в VxD xx".

Модуль VXDCALL.C, предоставляемый фирмой Vireo (изготовителем VTolsD) исправляет неправильный код во время выполнения. Скомпилируйте код однажды, и просто скомпонуйте в OBJ файл с любым VxD, созданным в VC ++ 4.1. Обратите внимание, что Вы должны также включить сопровождающий заголовочный файл VXDCALL.H во все исходные Си файлы вашего VxD.

Фирма Vireo предоставляет VXDCALL.C на странице [www.vireo.com](http://www.vireo.com). Файл VXDCALL.C нужен, если используется VC ++ 4.1, независимо от того, пользуетесь ли Вы DDK или VToolsD.

Файл SKELCTRL.ASM (листинг 4.2) обеспечивает создание блоков для SKELETON.VXD, а позднее и для VxD. SKELCTRL.ASM может быть легко адаптирован для использования в другом VxD путем изменения поля DDB (например, имя VxD) и добавления/удаления желаемых сообщений в *управляющей процедуре устройства*. Другой файл – SKELETON.C – содержит функции обработки сообщений, которые реализуют специфические функциональные возможности VxD, и существенно различны для различных VxD.

Хотя конкретные функциональные возможности исходного файла Си различны для каждого VxD, здесь и далее каждая версия исходного файла Си включает один и тот же основной набор заголовочных файлов. Заголовочные файлы с кратким описанием приведены в табл. 4.1.

Файл сборки (makefile) SKELETON.MAK (листинг 4.3) используется для создания SKELETON.VXD. Makefile компилирует, ассемблирует и связывает все компоненты, необходимые для создания SKELETON.VXD. После создания SKELETON.VXD, файл сборки (makefile) запускает утилиту MAPSYM, которая конвертирует (преобразует) map-файл компоновщика в символьный файл, годный к употреблению отладчиком (WDEB386 или SoftIce/Win).

Режимы работы (флаги) компилятора и ассемблера определены в макроопределениях CVXD\_FLAGS и AFLAGS в начале файла сборки (makefile). Таблицы 4.2 и 4.3 объясняют цель каждого из этих флагов.

Таблица 4.1.

## Заголовочные файлы для SKELETON.C

Заголовочный файл	Описание	Каталог
BASEDEF.H	Константы и типы, используемые другими заголовочными файлами	Inc32 в Win 95 DDK
DEBUG.H	Макроопределения для разрешения/запрещения кода отладки	Inc32 в Win 95 DDK
VMM.H	Константы и типы для сервисных функций VMM	Inc32 в Win 95 DDK
VXDWRAPS.H	Прототипы сервисных функций для VMM/VxD, предоставляемые VXDWRAPS.CLB	Inc32 в Win 95 DDK
WRAPPERS.H	Прототипы сервисных функций для VMM/VxD, предоставляемые WRAPPERS.CLB	wrappers
VXD_CALL.H	Исправления к прототипам функций VMMcall/VxDcall фирмы Vireo	wrappers
INTRINSI.H	Прототипы строковых функций для intrinsic	wrappers

Таблица 4.2.

## Параметры и флаги компилятора для VxD

Параметры и флаги	Цель
C	Только компиляция (без компоновки)
Gs	Запрещение проверки переполнения стека
Zdp, Zd	Имя PDB файла, который хранит отладочную и символьную информацию
Zl	Запрещение имени стандартной runtime библиотеки Си в OBJ; предотвращает случайную компоновку с неподдерживаемым runtime Си.
DIS_32	Определяет 32-, а не 16-разрядный код; используется некоторыми заголовочными файлами VxD.
DDEBUG	Разрешает отладку макроопределений и функций в некоторых заголовочных файлах VxD.
DDEBLEVEL=1	Устанавливает нормальный уровень отладки в DEBUG.H (возможные варианты выборочный <i>retail</i> , нормальный <i>normal</i> или максимальный <i>max</i> ).
DWANTVXDWRAPS	Запрещение некоторых <i>inline</i> -функций в заголовочных файлах VxD, вынуждая использовать вместо них функции из библиотеки оболочки

Таблица 4.3.

## Флаги ассемблера для VxD

Параметры и флаги	Цель
C	Только ассемблирование (без компоновки)
Coff	Выходной файл в формате COFF; теперь MS компоновщик использует файл COFF, а не OMF.
Cx	Сохранение регистра в переменных типа <i>publics</i> и <i>externs</i> .
W2	Установка 2 уровня предупреждений.
Zd	Включение номера строки в отладочную информацию OBJ.
DIS_32	Определяет 32-, а не 16-разрядный код; используется некоторыми заголовочными файлами VxD.
DDEBUG	Разрешает отладку макроопределений и функций в некоторых заголовочных файлах VxD.
DDEBLEVEL=1	Устанавливает нормальный уровень отладки в DEBUG.INC (возможные варианты выборочный <i>retail</i> , нормальный <i>normal</i> или максимальный <i>max</i> ).
DASM6	Определяется ассемблер MASM 6.x (используется некоторыми заголовочными файлами VxD)
DBLD_COFF	Определяет COFF формат (используется некоторыми заголовочными файлами VxD)

#### 4.3. Блок дескриптора устройства DDB и Управляющая Процедура Устройства: SKELCTRL.ASM

Короткий модуль на языке ассемблера SKELCTRL.ASM (листинг 3.2) содержит блок дескриптора устройства DDB и *управляющую процедуру устройства*:

```
.386p
;*****
;
;   Включения
;*****
;
;   include vmm.inc
;   include debug.inc
;=====
;   Объявление виртуального устройства
;=====
DECLARE_VIRTUAL_DEVICE  SKELETON, 1, 0, ControlProc,
UNDEFINED_DEVICE_ID, \
                        UNDEFINED_INIT_ORDER
VxD_LOCKED_CODE_SEG
;=====
;   Процедура: ControlProc
;
;   Описание:
;   Управляющая процедура устройства для SKELETON VxD
;   Вход:
;   EAX = Идентификатор вызова управления
;   Выход:
;   если флаг переноса очищен
;       Успешно
;   иначе
;       Вызов управления неудачен
;   Используются:
;   EAX, EBX, ECX, EDX, ESI, EDI, флаги
;=====
BeginProc ControlProc
    Control_Dispatch SYS_VM_INIT, _OnSysVmInit, cCall, <ebx>
    Control_Dispatch SYS_VM_TERMINATE, _OnSysVmTerminate, cCall, <ebx>
    Control_Dispatch CREATE_VM, _OnCreateVm, cCall, <ebx>
    Control_Dispatch DESTROY_VM, _OnDestroyVm, cCall, <ebx>
    Control_Dispatch CREATE_THREAD, _OnCreateThread, cCall, <edi>
    Control_Dispatch DESTROY_THREAD, _OnDestroyThread, cCall, <edi>
    cld
    ret
EndProc ControlProc
VxD_LOCKED_CODE_ENDS
END
```

В самом начале файла макроопределением `DECLARE_VIRTUAL_DEVICE_DDB` объявлен блок дескриптора устройства DDB. Параметры этого макроопределения один к одному соответствуют полям блока дескриптора устройства DDB, описанного в секции "Блок Дескриптора Устройства" главы 3. `SKELCTRL.ASM` использует только первые шесть макропараметров, потому что `VxD` не экспортирует API ни `V86`, ни защищенного режима. Поскольку `SKELETON` не экспортирует API или другие сервисные функции, не требуется `VxD ID`, так что `SKELCTRL.ASM` использует `UNDEFINED_DEVICE_ID` для макропараметра *Device\_Num* (*Device\_Num* – то же, что *Device ID*). `SKELETON` не требует специального порядка инициализации, так что для макропараметра *Init\_Order* используется значение `UNDEFINED_INIT_ORDER`.

Вторая половина `SKELCTRL.ASM` определяет *управляющую процедуру устройства VxD* (*Control Proc*). Управляющая процедура устройства `VxD` должна размещаться в заблокированном сегменте, для этого *Control Proc* окружен макроопределениями `VXD_LOCKED_CODE_SEG` и `VXD_LOCKED_CODE_ENDS`. *Control Proc* использует последовательность макроопределений *Control\_Dispatch* генерации кода основного оператора переключения. Например, строка

```
Contro1_Dispatch SYS_VM_INIT, _OnSysVmlnit, cCall. <ebx>
```

транслируется в код, который сравнивает код сообщения в `EAX` с `SYS_VM_INIT`, и, если они совпадают, вызывает функцию *OnSysVMInit* в модуле Си, принимая обработчик VM в `EBX` как параметр.

Этой информации о `SKELCTRL.ASM` достаточно чтобы модифицировать `VxD` для обработки других сообщения в имеющемся драйвере.

Остановимся более подробно на реальных функциональных возможностях SKELETON.VXD, содержащихся в SKELETON.C.

SKELETON.C (листинг 4.1) содержит обработчики сообщений для SKELETON.VXD. SKELETON.VXD обрабатывает шесть сообщений, касающихся создания и разрушения виртуальных машин и потоков: *Sys\_VM\_Init*, *Sys\_VM\_Terminate*, *Create\_VM*, *Destroy\_VM*, *Create\_Thread* и *Destroy\_Thread*. Каждый раз при создании VM всем VxD посылаются одно из двух сообщений: *Sys\_VM\_Init* для системной VM или *Create\_VM* для несистемных VM. Создание VM завершается созданием начального потока, но в этом случае не посылаются никакого сообщения. Последующие (не начальные) потоки, созданные в VM, кончаются сообщением *Create\_Thread*. Как уже говорилось ранее, каждая несистемная VM ограничивается единственным сообщением, поэтому все сообщения *Create\_Thread* связаны с системной VM.

SKELETON демонстрирует это поведение при печати из обоих обработчиков VM и значений обработчиков потоков для этих шести сообщений. Обработчики сообщений VM (*OnSysVmInit*, *OnCreateVm*, *OnDestroyVm*, и *OnSysVmTerminate*) используют сервисную функцию VMM *Get\_Initial\_Thread\_Handle*, чтобы получить обработчик начального сообщения, созданного вместе с VM. (Это сервисная функция не поддерживается библиотекой VXDWRAPS .CLB DDK, так что ее оболочка должна находиться в WRAPPERS.CLB). Обработчики сообщений потоков *Create\_Thread* и *Destroy\_Thread* извлекаются виртуальной машиной (VM), связанной с потоком, из обработчика потока, который указывает на реальный блок управления потоком. Одно из полей в блоке управления потоком – обработчик VM, связанной с потоком.

Каждая функция обработчика сообщения отправляет сообщения обработчика потока в отладчик и в файл. Функции используют

макроопределение `DPRINTF`, чтобы сгенерировать выход отладчика. Это макроопределение похоже на полезную функцию `VToolsD dprintf`. Макроопределение комбинирует запрос к двум сервисным функциям VMM: `_Sprintf`, которая форматирует строку, и `Out_Debug_String`, которая выводит форматированную строку в отладчик. Обе сервисные функции включены в библиотеку `VXDWRAPS.CLB DDK`.

Макроопределение разворачивается только в том случае, если во время трансляции обнаружен символ отладчика (`DEBUG`). Этот символ чаще определяется ключом компилятора, чем строкой `#define` в исходном файле. Например, с компилятором Microsoft используется `DDEBUG=1`. Если отладчик (`DEBUG`) не обнаружен, макроопределение `DPRINTF` вообще не разворачивается.

Чтобы посылать сообщения в файл, обработчики сообщений используют сервисную функцию `IFSMgr_Ring0_FileIO`. `IFSMgr` – это устанавливаемый менеджер файловой системы `VxD`, менеджер высшего уровня всех `VxD`, которые вместе формируют файловую систему. Большинство сервисных функций `IFSMgr` используются другими `VxD`, которые являются частью файловой системы, однако, сервисная функция `IFSMgr_Ring0_FileIO` полезна для любого `VxD`: она позволяет `VxD` исполнять файл ввода/вывода в Кольце 0. "Кольцо 0" – это важно, потому что, раньше `VxD` мог исполнять файл ввода/вывода только, переключаясь в кольцо 3, и каждая индивидуальная операция ввода/вывода (открытие, закрытие, и т.д.) требовала последовательности из нескольких сервисных функций VMM. Под Windows 95, требуется единственное обращение к `IFSMgr` для выполнения каждой файловой операции ввода/вывода.

Сервисная функция `IFSMgr_Ring0_FileIO` не будет работать правильно, если используется перед сообщением `Sys_Init_Complete`.

Хотя фактически сервисная функция *IFSMgr* использует единственную точку входа для всех операций ввода/вывода (открытие, закрытие, и т.д.) с кодом функции, их отличающим, более удобно иметь отдельную функцию для вызова каждой операции. При создании функций оболочки в WRAPPERS.CLB для каждого сообщения (*IFSMgr\_Ring0\_OpenCreateFile*, *IFSMgr\_Ring0\_WriteFile*, и т.д.) целесообразно использовать отдельную функцию, как это сделано в VToolsD.

Во время создания системной VM *OnSysVmInit* открывает файл VXDSKEL.LOG с вызовом *IFSMgr\_Ring0\_OpenCreateFile*. Интерфейс *IFSMgr\_Ring0\_OpenCreateFile* похож на интерфейс функции File Open INT 21h, с параметрами для имени файла, режима открытия (чтение, запись, и флаги разделения), атрибутов создания (нормальный, скрытый, и т.д.), и действия (чтобы не было сбоя, если файл не существует, и т.д.). Фактически, режим, атрибуты и параметры действия используют точно те же значения что и в INT 21h File Open (открытие файла).

*IFSMgr* добавляет два дополнительных параметра к запросу открытия (Open), которые не входят в интерфейс INT 21h. Один – двоичный контекст: если он установлен, файл открыт в контексте текущего потока, и к нему можно обращаться только, когда этот поток текущий. Другой параметр содержит флаговый бит, установка которого означает запрет кэширования чтения и записи для этого файла.

*OnSysVmInit* использует "создание и усечение" для параметра действия, так, чтобы файл регистрации был создан, если он не существует, или открывался и усекался, если существует. *OnSysVmInit* позволяет кэшировать файл (так как ввод/вывод в файл регистрации не критичен) и использует значение ЛОЖНО для двоичного контекста, так, чтобы VxD мог создавать файл ввода/вывода в любое время, независимо от того,

какой поток является текущим. Это позволяет VxD открывать файл во время *Sys\_VM\_Init*, когда начальный поток системной VM является текущим, а осуществлять запись в файл тем же самым обработчиком во время другого сообщения VM или потока, когда текущим является другой поток.

*OnSysVmInit* держит файл открытым и хранит дескриптор файла в глобальной переменной *fh* так, чтобы обработчики сообщений другого SKELETON.VXD могли также писать в файл. Файл закрывается обработчиком сообщения *OnSysVmTerminate*, при завершении работы Windows.

Все обработчики сообщений, включая *OnSysVmInit*, осуществляют запись в этот уже открытый файл, используя *IFSMgr\_Ring0\_WriteFile*. Эта функция использует параметры, которые обычно используются для записи: дескриптор, буфер и счетчик. Однако, если большая часть файловых функций ввода/вывода изменяет положение указателя файла автоматически с каждой операцией чтения и записи, *IFSMgr\_Ring0\_WriteFile* требует явного задания параметра положения указателя файла. Это означает, что вызывающий должен сохранять у себя текущее положение указателя файла. SKELETON делает это, обнуляя при инициализации глобальную переменную *file\_pos*, и инкрементируя ее количеством байтов, записанных с каждым обращением к *IFSMgr\_Ring0\_WriteFile*.

*IFSMgr\_Ring0\_WriteFile* не выполняет никакого форматирования, она просто пишет строку буфера. Перед вызовом *IFSMgr\_Ring0\_WriteFile* каждый обработчик сообщения сначала форматирует буфер, используя сервисную функцию VMM *\_Sprintf*, представленную в библиотеке VXDWRAPS.CLB DDK .



```

if (!fh)
  { DPRINTF(buf, "Error %x opening file %s\n", err, "vxdskel.log" );
  }
else
  { _Sprintf(buf, "SysVMInit: VM=%x tcb=%x\r\n", hVM, tcb );
    count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(buf), filepos, &err);
    filepos += count;
  }
return TRUE;
}

```

```

VOID OnSysVmTerminate(VMHANDLE hVM)
{ WORD err;
  int count=0;
  PTCB tcb;

  tcb = Get_Initial_Thread_Handle(hVM);
  DPRINTF( buf, "SysVmTerminate VM=%x tcb=%x\r\n", hVM, tcb );
  _Sprintf( buf, "SysVmTerminate VM=%x tcb=%x\r\n", hVM, tcb );
  count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(buf), filepos, &err);
  filepos += count;
  IFSMgr_Ring0_CloseFile( fh, &err );
}

```

```

BOOL OnCreateVm(VMHANDLE hVM)
{ PTCB tcb;
  WORD err;
  int count=0;

  tcb = Get_Initial_Thread_Handle(hVM);
  DPRINTF(buf, "Create_VM: VM=%x, tcb=%x\r\n", hVM, tcb);
  _Sprintf(buf, "Create_VM: VM=%x, tcb=%x\r\n", hVM, tcb);
  count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(buf), filepos, &err);
  filepos += count;
  return TRUE;
}

```

```

VOID OnDestroyVm(VMHANDLE hVM)
{ WORD err;
  int count;
  PTCB tcb;

  tcb = Get_Initial_Thread_Handle(hVM);
  DPRINTF(buf, "Destroy_VM: VM=%x tcb=%x\r\n", hVM, tcb );
  _Sprintf(buf, "Destroy_VM: VM=%x tcb=%x\r\n", hVM, tcb );
  count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(count), filepos, &err);
  filepos += count;
}

```

```
BOOL OnCreateThread(PTCB tcb)
```

```
{ WORD err;  
  int  count;
```

```
  DPRINTF(buf, "Create_Thread: VM=%x, tcb=%x\r\n", tcb->TCB_VMHandle, tcb);  
  _Sprintf(buf, "Create_Thread: VM=%x, tcb=%x\r\n", tcb->TCB_VMHandle, tcb);  
  count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(count), filepos, &err);  
  filepos += count;  
  return TRUE;
```

```
}
```

```
VOID OnDestroyThread(PTCB tcb)
```

```
{ WORD err;  
  int  count;
```

```
  DPRINTF( buf, "Destroy_Thread VM=%x, tcb=%x\r\n", tcb->TCB_VMHandle, tcb );  
  _Sprintf( buf, "Destroy_Thread VM=%x, tcb=%x\r\n", tcb->TCB_VMHandle, tcb );  
  count = IFSMgr_Ring0_WriteFile(FALSE, fh, buf, strlen(count), filepos, &err);  
  filepos += count;
```

```
}
```

## Листинг 3.2. SKELCTRL.ASM (версия DDK)

```

.386p
;*****
;
;   INCLUDES
;*****
;
;   include vmm.inc
;   include debug.inc
;=====
;   VIRTUAL DEVICE DECLARATION
;=====
DECLARE_VIRTUAL_DEVICE  SKELETON, 1, 0, ControlProc,
UNDEFINED_DEVICE_ID, \
    UNDEFINED_INIT_ORDER
VxD_LOCKED_CODE_SEG
;=====
;   ПРОЦЕДУРА: ControlProc
;
;   ОПИСАНИЕ:
;   Device control procedure for the SKELETON VxD
;
;   ВХОД:
;   EAX = Control call ID
;
;   ВЫХОД:
;   если флаг переноса сброшен
;     Успешно
;   иначе
;     Вызов неудачен
;
;   ИСПОЛЬЗУЕТ:
;   EAX, EBX, ECX, EDX, ESI, EDI, Flags
;=====

BeginProc ControlProc
    Control_Dispatch SYS_VM_INIT, _OnSysVmInit, cCall, <ebx>
    Control_Dispatch SYS_VM_TERMINATE, _OnSysVmTerminate, cCall, <ebx>
    Control_Dispatch CREATE_VM, _OnCreateVm, cCall, <ebx>
    Control_Dispatch DESTROY_VM, _OnDestroyVm, cCall, <ebx>
    Control_Dispatch CREATE_THREAD, _OnCreateThread, cCall, <edi>
    Control_Dispatch DESTROY_THREAD, _OnDestroyThread, cCall, <edi>

    cld
    ret

EndProc ControlProc

VxD_LOCKED_CODE_ENDS

    END

```

*Листинг 3.3. SKELETON.МАК (версия DDK)*

```

CVXDFLAGS = -Zdp -Gs -c -DIS_32 -ZI -DDEBLEVEL=1 -DDEBUG -
DWANTVXDWRAPS=1
AFLAGS    = -coff -DBLD_COFF -DIS_32 -W2 -Zd -c -Cx -DMASM6 -
DDEBLEVEL=1 -DDEBUG

all:       skeleton.vxd

skeleton.obj: skeleton.c
            cl $(CVXDFLAGS) -Fo$@ -Fc %s

skelctrl.obj: skelctrl.asm
              ml $(AFLAGS) -Fo$@ %s

skeleton.vxd: skelctrl.obj skeleton.obj ..\..\wrappers\vxdcall.obj
..\..\wrappers\wrappers.clb skeleton.def
            echo >NUL @<<skeleton.crf
-MACHINE:i386 -DEBUG -DEBUGTYPE:MAP -PDB:NONE
-DEF:skeleton.def -OUT:skeleton.vxd -MAP:skeleton.map
-VXD vxdwraps.clb wrappers.clb skelctrl.obj skeleton.obj vxdcall.obj
<<
            link @skeleton.crf
            mapsym skeleton

```

## Листинг 3.4. SKELETON.DEF (версия DDK)

```

VXD SKELETON
SEGMENTS
  _LTEXT      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _LDATA      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _TEXT       CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _DATA       CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _LPTEXT     CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _CONST      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _BSS        CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _TLS        CLASS 'LCODE'  PRELOAD NONDISCARDABLE
  _ITEXT      CLASS 'ICODE'  DISCARDABLE
  _IDATA      CLASS 'ICODE'  DISCARDABLE
  _PTEXT      CLASS 'PCODE'  NONDISCARDABLE
  _PDATA      CLASS 'PCODE'  NONDISCARDABLE
  _STEXT      CLASS 'SCODE'  RESIDENT
  _SDATA      CLASS 'SCODE'  RESIDENT
  _MSGTABLE   CLASS 'MCODE'  PRELOAD NONDISCARDABLE IOPL
  _MSGDATA    CLASS 'MCODE'  PRELOAD NONDISCARDABLE IOPL
  _IMSGTABLE  CLASS 'MCODE'  PRELOAD DISCARDABLE IOPL
  _IMSGDATA   CLASS 'MCODE'  PRELOAD DISCARDABLE IOPL
  _DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
  _DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
  _DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
  _16ICODE    CLASS '16ICODE' PRELOAD DISCARDABLE
  _RCODE      CLASS 'RCODE'
EXPORTS
  SKELETON_DDB @1

```

#### 4.4. Резюме

SKELETON.VXD, даже с его ограниченными функциональными возможностями, иллюстрирует множество проблем возникающих при разработке VxD, требуя правильного использования структур, интерфейса, и инструментальных средств. Используя библиотеку оболочек WRAPPERS.CLB, можно писать виртуальные драйверы VxD для Windows 95 непосредственно на Си, даже если Вы имеете только средства API DDK.

#### 4.5. Контрольные вопросы

1. Чем отличается процесс написания VxD с использованием средств DDK Windows 95 и пакета VToolsD?
2. Для чего нужны ассемблерные фрагменты при написании VxD?
3. Почему VxD нельзя отлаживать со стандартными отладчиками?
4. Что содержит файл SKELCTRL.ASM?
5. Для чего нужна *управляющая процедура устройства*?
6. Для чего нужен файл сборки?
7. Что определяют флаги компилятора?
8. Что определяют флаги ассемблера?
9. Для чего нужен блок дескриптора устройства?
10. Что означает макроопределение VXD\_LOCKED\_CODE\_SEG?
11. Почему управляющая процедура устройства располагается в заблокированном сегменте?
12. Что означает макроопределение VXD\_LOCKED\_CODE\_ENDS?
13. В каком случае вызывается функция *OnSysVMInit*?
14. Какие сообщения обрабатывает SKELETON.VxD?
15. Какое сообщение посылается VxD при создании виртуальной машины?

16. Какое сообщение посылается VxD при создании начального потока виртуальной машины?
17. С какой виртуальной машиной связаны сообщения *Create\_Thread*, посылаемые VxD?
18. Для чего нужна сервисная функция *Get\_Initial\_Thread\_Handle*?
19. Как передаются обработчики сообщений потоков *Create\_Thread* и *Destroy\_Thread*?
20. Что такое *IFSMgr*?
21. Для чего нужна сервисная функция *IFSMgr\_Ring0\_FileIO*?
22. Каково условие правильной работы функции *IFSMgr\_Ring0\_FileIO*?
23. Для чего нужна сервисная функция *IFSMgr\_Ring0\_OpenCreateFile*?
24. Чем отличается функция *IFSMgr\_Ring0\_OpenCreateFile* от соответствующей функции DOS?
25. Для чего нужна сервисная функция *IFSMgr\_Ring0\_WriteFile*?

*Литература*

1. Адриан Кинг. Windows 95 изнутри/Перев. с англ. – СПб: Питер, 1995. – 512 с.: ил.
2. Мюррей У., Паппас К. Создание переносимых приложений для Windows/ Пер. с англ. – СПб.: BHV – Санкт-Петербург, 1997. – 816 с.: ил.
3. Ресурсы Microsoft Windows 95: В 2 т. Т.1./Пер. с англ. – М.: Издательский отдел “Русская редакция” ТОО “Channel Trading Ltd”, 1996. – 424 с.: ил.
4. Ресурсы Microsoft Windows 95: В 2 т. Т.2./Пер. с англ. – М.: Издательский отдел “Русская редакция” ТОО “Channel Trading Ltd”, 1996. – 424 с.: ил.
5. Karen Hazzah/ Writing Windows VxDs and Device Drivers/ – R&D Books, Lawrence, KS 66046, 1997. – 480 s.

*Учебное пособие*

*Рощин Алексей Васильевич*

Организация ввода-вывода.

Часть 1. Виртуальные драйверы и виртуальное окружение WINDOWS.  
Учебное пособие.

---

Подписано к печати 18.04.2002 г.

Формат 60×84 1/16.

Объем 5,0 п.л. Тираж 300 экз. Заказ № 62

Московский государственный университет  
приборостроения и информатики  
107076, Москва, ул. Стромьнка 20.