

Материалы к занятию 15 апреля 2020

Тема: простейший драйвер.

Загружаемый драйвер **DOS** – это резидент, имеющий специальную структуру, и выполняющий заранее заданный перечень команд (см. пособие). Все обращения к драйверу выполняются операционной системой (**DOS**). Это же касается драйверов под **Windows**. Хотя в прикладной программе **Windows**, вроде бы, можно обратиться к драйверу (например, последовательный порт открывается, как файл с конкретным дескриптором и параметрами). На самом деле, **WinAPI** транслирует это обращение в системный вызов, и опять-таки к драйверу обращается операционная система.

Важно знать: драйверы образуют в **DOS** связный список (то есть, каждый предыдущий драйвер имеет в заголовке поле «Адрес следующего устройства», в котором записан адрес (полный – **сегмент:смещение**) начала следующего в списке драйвера. Так как после последнего драйвера ничего нет, у него в это поле записано значение **0fffffffh** (или, что то же самое, **-1**). То есть, значение **0fffffffh** – признак последнего драйвера в цепочке (списке). В качестве первого драйвера списка всегда выступает стандартный драйвер **DOS** под названием **nul**, который, в отличие от любого досового драйвера, нельзя заменить.

Отметим важные отличия драйверов от резидентных программ, описанных ранее:

- если резидентная программа, это абсолютно произвольный код (за который полную ответственность несет его автор), то драйвер имеет строго заданную структуру (с обязательным 18-байтовым заголовком драйвера),
- проблемы коммуникации (вызова, передачи данных и т. д.) с резидентной программой решаются ее автором произвольным образом, в то время, как драйвер может вызываться только операционной системой, поэтому интерфейс вызова жестко задан (для каждой команды задан формат данных, которые передаются драйверу или принимаются от него),
- все команды, которые может обрабатывать драйвер конкретного типа, должны обрабатываться (если какие-то команды данным драйвером не обрабатываются, драйвер должен зафиксировать это в своем ответе – состоянии, **rh_status**),
- при любой обработке любой команды в слове состояния должен быть взведен атрибут «сделано» (**100h**), только в этом случае система поймет, что драйвер сработал,
- любое обращение системы к драйверу состоит из двух шагов – вызов процедуры **Стратегия**, и сразу после этого вызов процедуры **Прерывание**; процедура **Стратегия** вызывается исключительно для занесения адреса заголовка запроса (содержащегося в паре регистров **ES:BX**) текущей команды во внутренние переменные драйвера (бывают и дополнительные действия в этой процедуре, но

крайне редко); процедура **Прерывание** занимается собственно обработкой команд,

- если резидентная программа может быть загружена в любой момент текущего сеанса работы (а если она написана соответствующим образом, то и выгружена), то драйвер может быть загружен только при загрузке **DOS** (статический драйвер; в **Windows** драйверы могут быть как статическими, так и динамическими, то есть, загружаемыми и выгружаемыми в любое время), а прекращает свое существование (выгружается) драйвер только с выгрузкой системы,
- накладные расходы резидентной программы определяются наличием **PSP** (256 байтов), окружением (**Enviriment** – минимум 178 байтов) и стеком; накладные расходы драйвера – только его заголовок (18 байтов), так как стеком он пользуется системным.

Как видите, отличия существенные.

В этом занятии мы напишем простейший драйвер, который выполняет единственную команду (0 – Инициализация), в которой перехватывается прерывание **int 5** (реакция на клавишу **PrtSc**). Новая процедура обработки прерывания **int 5** по каждому нажатию клавиши **PrtSc** выводит на экран символ '#'. Затем подробно разберем его трансляцию и превращение в собственно драйвер, а также загрузку его при старте операционной системы.

ВАЖНО. Загрузить драйвер и проверить его работу можно только под «чистой» **DOS** (не в сеансе и не **DosBox**'е) или в виртуальной машине (виртуальные диски для обеих виртуальных машин уже содержат все необходимое для загрузки драйвера при старте **DOS** – см. файл **config.sys** в корне диска).

Напишем также резидентную программу, которая делает то же самое (перехватывает прерывание **int 5**), но при нажатии на клавишу **PrtSc** выводит символ '\$'. Запустим ее, и убедимся, используя программу **mi.com**, что драйвер занимает существенно меньше места, чем соответствующая резидентная программа.

Все, о чем сегодня идет речь, находится в папке **04_15N** в общем архиве.

Итак, простейший драйвер **drv.asm** (листинг ниже).

Начало текста (строки 4 и 5 – **code segment** и **.286**) пояснений, очевидно, не требуют.

Дальше идут определения структур заголовков запросов (так странно называются пакеты данных, отправляемые драйверу для выполнения каждой команды **rh** – **Request Header**). Эти определения являются инструкциями, предназначенными только для компилятора. Они лишь облегчают работу программиста. Так адрес, описанный в строке 39 (**es:[bx].rh_cmd**), означающий, что надо взять поле **cmd** из заголовка запроса без определения структуры пришлось бы записывать так: **byte ptr es:[bx+2]**. То есть, программист без определения структур должен следить за форматом каждого поля (байт, слов и т. д.), а также за его расположением в заголовке запроса (+2). Компилятор использует описанные структуры для того, чтобы

подставить в команду конкретные формат и адрес (**byte ptr es:[bx+2]**). После компиляции все определения структур исключаются, так как они уже не нужны.

Теперь, собственно, о структурах (подробнее здесь: <https://works.doklad.ru/view/Do4ks1utDvI.html>). Далее прямая цитата оттуда: «Описание шаблона структуры имеет следующий синтаксис:

```
имя_структуры STRUC  
<описание полей>  
имя_структуры ENDS
```

Здесь <описание полей> представляет собой последовательность директив описания данных db, dw, dd, dq и dt.

Их операнды определяют размер полей и, при необходимости, начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Как мы уже отметили при описании шаблона, память не выделяется, так как это всего лишь информация для транслятора.

Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начале сегмента данных либо перед ним.»

Еще одна цитата:

«Идея введения структурного типа в любой язык программирования состоит в объединении разнотипных переменных в один объект.

В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор — символ "." (точка). Он используется в следующей синтаксической конструкции:

адресное_выражение.имя_поля_структуры

Здесь:

- адресное_выражение — идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ниже синтаксическими правилами (рисунок 1);
- имя_поля_структуры — имя поля из шаблона структуры.

Это, на самом деле, тоже адрес, а точнее, смещение поля от начала структуры.

Таким образом оператор "." (точка) вычисляет выражение

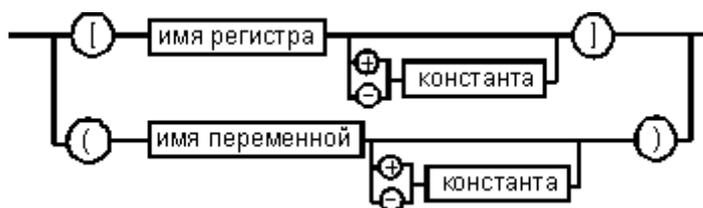


Рисунок 1 – Синтаксис адресного выражения в операторе обращения к полю структуры»

Следующая цитата оттуда же понадобится нам в следующем занятии:

«Определение данных с типом структуры

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция:

[имя переменной] имя_структуры <[список значений]>

Здесь:

- имя переменной — идентификатор переменной данного структурного типа. Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры.
- список значений — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы.

Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с типом данной структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать угловые скобки.»

Итак, вернемся к нашим структурам. Как уже говорилось, для каждой команды имеется собственная структура передаваемого пакета данных (заголовка запроса). Однако, есть данные, которые в каждой команде необходимы и одинаковы. Это называется «фиксированная структура заголовка», и задается она структурой **rh** (просто заголовок запроса).

В этой структуре три байтовых поля, которые передаются от системы драйверу:

- **rh_len** – полная длина пакета (смещение от начала пакета 0),
- **rh_unit** – номер устройства (смещение +1), которое важно только для блочных устройств,
- **rh_cmd** – номер команды (смещение +2),

одно поле словное, передаваемое от драйвера системе:

- **rh_status** – состояние выполнение команды (смещение +3),

четырёхсловное поле – резерв, который так любит везде использовать фирма Microsoft:

- **rh_res.**

В нашем простейшем драйвере используется еще лишь одна структура, соответствующая команде 0 (Инициализация), так как другие команды этот драйвер не обрабатывает. Эта структура называется **rh0** (заголовок запроса команды 0 - ноль):

- **rh0_rh db size rh dup(?)** – это поле включает в себя все поля, описанные в предыдущей структуре (занимает 13 байтов), поэтому здесь не подробно описывается и называется фиксированная часть,
- **rh0_nunits** – байтовое поле **Число устройств в группе** (смещение +13), которое важно только для блочных устройств,
- **rh0_brk_ofs** – словное поле **Смещение конца драйвера** (смещение в заголовке запроса +14), определяющее смещение адреса конца драйвера,
- **rh0_brk_seg** – словное поле **Сегмент конца драйвера** (смещение в заголовке запроса +16) , определяющее сегмент адреса конца драйвера,
- **rh0_bpb_tbo** – словное поле **Смещение указателя массива BPB** (смещение в заголовке запроса +18),
- **rh0_bpb_tbs** – словное поле **Сегмент указателя массива BPB** (смещение в заголовке запроса +20),
- **rh0_drv_itr** – байтовое поле **Первый доступный накопитель** (смещение в заголовке запроса +22).

Последние три поля также важны только для блочных устройств. Здесь **BPB** – Блок Параметров BIOS (**BIOS Parameter Block**) – это набор данных, характеризующий данный конкретный накопитель, и содержащийся в корневом секторе. Более подробно это описано в пособии при описании драйвера RAM диска (с. 153, 158).

Отметим, что драйверы, описанные в пособии (драйвер RAM диска – блочный и драйвер консоли – символьный) вполне работоспособны. В них определены структуры всех команд, обрабатываемых драйверами.

Отметим также, что определение структуры завершается высказыванием **имя_структуры ENDS**, где ключевое слово **ENDS** совпадает с тем, что пишется при завершении сегмента. Это не страшно, так как компилятор по контексту поймет, относится это к структуре, или к сегменту.

Строка 24 (**org 0**) говорит о том, что компиляция начинается с адреса нуль, как это обычно принято для программ типа **.exe** (а драйвер, как мы укажем позднее, транслируется как программа такого типа, чтобы не резервировать балластные 256 байтов для **PSP**). Эту строку можно было не писать, так как при отсутствии инструкции **org** компиляция по умолчанию начинается с нуля. Здесь она указана для того, чтобы Вы обратили на это внимание.

Строка 25 (**simple proc far**) определение дальней (так как вызов извне) основной процедуры.

Строки 28-32 – единственные накладные расходы драйвера, его заголовков. По-порядку.

Строка 28 (**next_dev dd -1**) поле адреса следующего устройства. При написании драйвера в это поле записывается **-1 (0fffffffh)**. Так что, если этот драйвер останется последним, это значение в поле сохранится. Если же после него будет загружаться еще драйвер, **DOS** заменит это значение на

действительный адрес следующего драйвера (который она знает, так как сама его загружает).

Строка 29 (**attribute dw 8000h**) означает, что данный драйвер является символьным (пособие, страница 150, таблица 5.1). Все остальные атрибуты нулевые, так как этот драйвер ничего больше (кроме загрузки и выполнения команды 0) делать не умеет.

Строка 30 (**strategy dw dev_strategy**) адрес (только смещение) процедуры **Стратегия**. Тот факт, что адрес процедуры описывается только смещением, говорит о том, что размер драйвера не может превышать одного сегмента (64 Кб).

Строка 31 (**interrupt dw dev_int**) такой же адрес процедуры **Прерывание** – основной процедуры драйвера.

Строка 32 (**dev_name db 'SIMPLE '**) – последнее поле заголовка драйвера – его имя. Так как драйвер у нас символьный, здесь должно быть имя драйвера длиной равной 8 символов. Так как наше имя **SIMPLE** содержит только 6 символов, мы должны дополнить его двумя пробелами (что и делаем). Напомню, что если бы драйвер был блочным, в поле **имя** действителен только первый байт, в котором указывается количество обслуживаемых драйвером устройств.

В строке 34 (**sym equ '#'**) мы определяем константу (выводимый по клавише **PrtSc** символ).

В строке 36 (**dev_strategy: retf**) находится вся процедура **Стратегия**. Мы говорили о том, что главная задача процедуры **Стратегия** заключается в сохранении во внутренних переменных драйвера адреса заголовка запроса, передаваемых **DOS** в регистрах **ES:BX**. Так как драйвер у нас чрезвычайно простой, и мы не собираемся портить регистры **ES:BX** (заметим, что по этой же причине в нашем драйвере не предусмотрены внутренние переменные для хранения этого адреса, которые являются необходимой принадлежностью любого серьезного драйвера). Таким образом, наша процедура **Стратегия** ничего не делает, только возвращает управление.

В строке 38 (**dev_int: pusha**) начинается процедура **Прерывание**. Обратим внимание на то, что хотя обе процедуры называются процедурами, формально мы их как процедуры не оформляем (ни к чему). Начинается процедура **Прерывание** с сохранения всех регистров общего назначения в стеке (напоминаю о необходимости возвращаться из любого резидентного кода с неизменными регистрами процессора).

Далее (строка 39 – **mov al,es:[bx].rh_cmd**) мы берем номер команды в регистр **AL**. Напоминаю, что **ES:[BX]** – адрес начала заголовка запроса, который передала драйверу **DOS**, а **.rh_cmd** – означает соответствующее поле заголовка запроса (в целом эта запись полностью эквивалентна **byte ptr es:[bx+2]**, но без лишних хлопот).

Так как наш драйвер обрабатывает единственную команду (нулевую), то у нас отсутствует формирование таблицы переходов по номеру команды (которая есть в обоих драйверах в пособии). Мы лишь проверяем, наша ли эта команда (строка 40 – **cmp al,0**).

Если в **AL** не нуль, в строке 41 (**jnz exit3**) выполняется условный (по нулю) переход на метку, где в поле статуса добавляется атрибут «неизвестная команда». Если команда нулевая, переход не выполняется, и мы переходим к строке 42 (**call init**), где вызывается процедура выполнения нулевой команды (инициализации).

Затем в строках 45-53 выполняется стандартный выход из драйвера.

Так как мы будем в поле статуса набирать атрибуты (с помощью команды **ИЛИ**), в строке 45 (**mov es:[bx].rh_status,0**) производится обнуление этого поля. Использовать экономную команду обнуления (**xor**) для этой цели мы не можем, так как обнуление выполняется в памяти, а оба операнда в памяти быть не могут.

В строках 46-48 в соответствующие поля записывается полный адрес конца драйвера. Мы под концом драйвера будем понимать начало процедуры инициализации, так как она выполняется только в момент загрузки драйвера (примерно так же, как и в резидентной программе).

В строке 46 (**mov es:[bx].rh0_brk_ofs,offset init**) в соответствующее поле заголовка запроса записывается смещение метки начала процедуры **init**. Обратите внимание на ключевое слово **offset**, которое определяет, что в поле будет загружено именно смещение этой метки. Адресация поля смещения конца драйвера (**es:[bx].rh0_brk_ofs**) формируется таким же образом, как и ранее. **es:[bx]** указывает на начало заголовка запроса, а **.rh0_brk_ofs** – на смещение поля относительно этого начала.

В строке 47 (**mov es:[bx].rh0_brk_seg,cs**) таким же образом в соответствующее поле записывается сегмент конца драйвера.

Так как на данном этапе команда выполнена правильно, в строке 48 (**jmp exit2**) переходим на метку, где будет установлен соответствующий атрибут.

В строке 49 (**exit3: or es:[bx].rh_status,8003h**) расположена метка **exit3:**, на которую мы переходим, если команда не нулевая (не обрабатываемая данным драйвером). Здесь с помощью команды **ИЛИ** мы взводим атрибут наличия ошибки (**8000h**), а в разрядах 0-7 записывается код ошибки (подробности можно посмотреть здесь: http://www.xserver.ru/computer/os/msdos/1/6.shtml#ch6_2). В нашем случае это код 3 (неизвестная данному драйверу команда), так как нашему драйверу все команды, кроме нулевой, неизвестны.

После этого в строке 50 (**exit2: or es:[bx].rh_status,100h**) таким же образом устанавливается бит «сделано», который должен устанавливаться всегда, когда команда как-то обрабатывалась (в данном случае она обработалась так, что опознана неизвестной). Кроме того, в этой строке находится метка **exit2:**, на которую выполняется переход при успешном выполнении команды (при этом строка 49 пропускается, и биты ошибки не устанавливаются).

Затем, в строке 51 (**popa**) восстанавливаются из стека все регистры общего назначения, и в строке 52 (**retf**) осуществляется выход из процедуры.

Далее в тексте программы описана новая процедура обработки прерывания **int 5** (строка 55 – **new_int5 proc far**).

Как всегда в резидентной процедуре сохраняем в стеке (строка 56 – **pusha**) и восстанавливаем из него (строка 60 – **popa**) все регистры общего назначения.

Собственно содержимое процедуры уложилось в три строки (57-59). Это вывод ранее определенного символа (#) в режиме телетайпа:

- строка 57 – **mov ah,0eh** (функция вывода в режиме телетайпа видеопрерывания **int 10h**),
- строка 58 – **mov al,sym** (записи выводимого символа в регистр **AL**),
- строка 59 – **int 10h** (вызов собственно видеопрерывания).

Стандартное замыкание процедуры обработки прерывания:

- строка 61 – **iret** (выход из процедуры обработки прерывания),
- строка 62 – **new_int5 endp** (замыкание формальной процедуры).

Наконец, в строке 63 начинается процедура инициализации (**init: push ds**) с сохранения в стеке сегментного регистра **DS**, так как мы будем его «портить». Нам необходимо его «испортить», так как **25h** функция системного прерывания (установка вектора прерывания) требует указания сегментного адреса устанавливаемой процедуры именно в регистре **DS**. Обратите внимание на то, что мы не сохраняем старый вектор прерывания **int 5**, так как нам не придется его восстанавливать. (Подумайте, почему.)

Строки 64 (**mov ax,cs**) и 65 (**mov ds,ax**) переписывают значение сегмента из **CS** (где мы находимся) в **DS** (что нужно для функции **25h**). Пересылка идет через промежуточный регистр (**AX**), так как прямая пересылка из сегментного регистра в другой сегментный запрещена.

В строке 67 (**mov ax,2505h**) задается функция **25h** (**AH=25h**) вектора прерывания **int 5** (**AL=5**).

В строке 68 в регистр **DX** загружается смещение новой процедуры обработки прерывания **int 5** (о сегментном регистре **DS** мы уже позаботились).

Наконец, в строке 69 вызывается системное прерывание **int 21h**, которое и устанавливает указанный вектор.

В строке 70 (**pop ds**) восстанавливается из стека сегментный регистр **DS**.

Наконец, в строке 71 (**ret**) – возврат из процедуры.

Строки 72 (**code ends**) и 73 (**end simple**) – стандартное завершение любой программы.

Для получения и установки драйвера необходимо:

1. Скомпилировать его:

tasm drv.asm – получится **drv.obj**

2. Скомпоновать:

tlink drv.obj – получится **drv.exe**

3. Отсечь заголовок **EXE** программы (описание и листинг этой программы приведен после листинга драйвера):

cut512 drv.exe - получится **drv.bin**

4. Поместить **drv.bin** в корень загрузочного диска **DOS**

5. Добавить в файл **config.sys** строку:

device=drv.bin

6. Перезапустить DOS

7. Попробовать понажимать клавишу **PrtScr**, и

убедиться, что на экран выводится символ **#**

Запустите программу **mi.com**, которая покажет расположение драйвера в памяти. Убедиться, что драйвер **drv** занимает в памяти 80 байтов, посмотрите, каковы атрибуты драйвера (совпадают ли они с теми, что прописаны в заголовке драйвера (**attribute dw 8000h**) и каково имя (**dev_name db 'SIMPLE '**).

Что касается экспериментов с драйвером, то можно, пожалуй, изменить символ, который будет выводиться по нажатию клавиши **PrtSc**. При этом стоит дать файлу драйвера другое имя (имя самого драйвера можно оставить неизменным), а затем заново скомпилировать и установить драйвер, как это описано выше. Следует только помнить, что неработающий драйвер может не дать загрузиться операционной системе (это касается и виртуальной машины). Поэтому имеет смысл располагать драйвер (двоичный, который загружается) не на системном диске (например, на внешней флэшке). Тогда при неудачной попытке загрузить драйвер можно вынуть флэшку. В этом случае при загрузке будет сообщение **DOS** о том, что ошибка в строке такой-то файла **config.sys**. Система при этом загрузится без драйвера.

На виртуальной машине в качестве другого носителя может выступать дополнительный жесткий диск.

Создание дополнительного жесткого диска на **Microsoft Virtual PC**:

1 Запустить виртуальную машину

2 В **Virtual PC Console** нажать **Settings**

3 Выбрать **Hard Disk 2**

4 Нажать **Virtual Disk Wizard**

Затем **Next**

Согласиться с **Create a new virtual disk (Next)**

Согласиться с **A virtual hard disk (Next)**

Нажать **Browse...** и выбрать место, где будет находиться этот диск (например, **D:\Temp**), ввести желаемое имя файла (например, **VD2.vhd** – расширение должно остаться **vhd**) и нажать **"Сохранить"**, затем нажать **(Next)**

5 Выбрать вариант **Fixed size (Next)**

6 Выбрать размер 5 Мб **(Next)**

7 Нажать **Finish** (и закрыть появившееся окно)

8 Выбрать **Virtual hard disk file**:

нажать **Browse...**

Найти файл, который создали выше (**D:\Temp\VD2.vhd**),
выбрать его и нажать кнопку "**Открыть**"

9 Теперь нажать кнопку "**OK**"

10 Нажать "**Старт**" на **VM DOS** (Убедиться, что диск **D** не появился).
Чтобы он появился на нем надо создать раздел и тформатировать.

11 Пройти в каталог **C:\DOS**

12 Запустить утилиту **fdisk**

Выбрать пункт 5 (просто нажать **5** и **Enter**)

Выбрать диск 2 (нажать **2** и **Enter**)

Выбрать пункт 1 (просто нажать **5** и **Enter**)

Выбрать пункт 1 **Create primary DOS Partition** (просто нажать **5** и **Enter**)

Согласиться с отведением всего пространства под раздел
(просто нажать **5** и **Enter**)

Нажать **Esc**

Еще раз нажать **Esc** и нажать любую кнопку

После перезагрузки **DOS** опять Пройти в каталог **C:\DOS**

13 Набрать **format D:**

14 Нажать **Y** и **Enter**

15 Отказаться и метки тома (нажать **Enter**)

16 Убедиться, что появился диск **D**

17 Перепишите файл **drv.bin** в корень диска **D:**

18 исправьте в файле **config.sys** строку **device=c:\drv.bin** на
device=d:\drv.bin

19 Перезагрузите **DOS** (Нажав "**Action**" и выбрав **Ctrl+Alt+Del**)

20 Запустите программу **mi.com** в папке **20_04_22** и убедитесь (нажав **Ctrl+O**), что драйвер загрузился

Создание дополнительного жесткого диска на **Oracle VM VirtualBox**:

1 Запустить виртуальную машину

2 В **Oracle VM VirtualBox Менеджер** нажать желтую шестеренку
(**Настроить**)

3 В левом столбце выбрать **Носители**

4 В окошке **Носители информации** выбрать контроллер: **IDE** и
нажать самую правую кнопку в этой строке (в подсказке – **Добавляет
жесткий диск**)

5 В открывшемся окне выбрать кнопку "**Создать новый диск**"

6 В новом окне выбрать пункт **VHD (Virtual Hard Disk)** + кнопку
"**Вперед**"

7 В новом окне выбрать пункт "**Фиксированный виртуальный
жесткий диск**" + кнопку "**Вперед**"

8 В следующем окне указать размер диска (самый маленький – **4,00
Мб**) + кнопку "**Создать**"

9 Нажать кнопку "**OK**"

10 На **VM DOS** нажать зеленую кнопку "**Запустить**" (Убедиться, что диск **D** не появился). Чтобы он появился на нем надо создать раздел и отформатировать.

Далее пункты с 11 по 20 из инструкции для **Microsoft Virtual PC**.

Листинг драйвера **drv.asm**.

```
;Драйвер, перехватывающий прерывания int 5
;и не делающий больше ничего
;По нажатию PrtSc на экран выводится '#'
code segment
    .286
; Структура заголовков запросов
rh struct ; Фиксированная структура заголовка
rh_len db ? ; Длина пакета (+0)
rh_unit db ? ; Номер устройства (+1)
rh_cmd db ? ; Команда (+2)
rh_status dw ? ; Возвращается драйвером (+3)
rh_res dq ? ; Резерв (+5)
rh ends
;Инициализация
rh0 struct ; Заголовок запроса команды 0
rh0_rh db size rh dup(?) ; Фикс.часть (13 байтов)
rh0_nunits db ? ; Число устройств в группе (+13)
rh0_brk_ofs dw ? ; Смещение конца драйвера (+14)
rh0_brk_seg dw ? ; Сегмент конца драйвера (+16)
rh0_bpb_tbo dw ? ; Смещ.указателя массива BPB (+18)
rh0_bpb_tbs dw ? ; Сегм.указателя массива BPB (+20)
rh0_drv_ltr db ? ; Первый доступный накопитель (+22)
rh0 ends
    org 0
simple proc far
    assume cs:code, ds:code
;Заголовок драйвера
next_dev dd -1 ;Адр.след.устр.0fffffffh
attribute dw 8000h ;Символьный
strategy dw dev_strategy ;Проц.Стратегия
interrupt dw dev_int ;Проц.Прерывание
dev_name db 'SIMPLE ' ;Имя устр.(8 символов)
;***** Конец заголовка драйвера
sym equ '#' ;Выводимый символ
;Процедура стратегия
dev_strategy: retf ;Ничего не делает
;Процедура прерывание
dev_int: pusha
    mov al,es:[bx].rh_cmd; Команда
    cmp al,0 ;Команда 0?
    jnz exit3;Нет
    call init ;Да
; Предполагается, что других вызовов этого драйвера не будет
; Запись в заголовок запроса адреса конца драйвера
    mov es:[bx].rh_status,0 ;Обнуление статуса
    mov es:[bx].rh0_brk_ofs,offset init;Смещ.конца
```

```

        mov     es:[bx].rh0_brk_seg,cs ;Сегмент конца
        jmp     exit2 ;Все в порядке
exit3: or     es:[bx].rh_status,8003h ; Неизвестная команда
exit2: or     es:[bx].rh_status,100h ; Сделано без ошибки
        popa
        retf
simple endp
; Новый обработчик int 5
new_int5 proc far
        pusha
        mov ah,0eh ;Вывод в реж.тетелайпа
        mov al,sym ;Символ
        int 10h ;Видеоперывание
        popa
        iret
new_int5 endp
init: push ds ;Сохранение DS в стеке
        mov ax,cs ;DS=CS для функ.25h int 21h
        mov ds,ax ;(это не COM прогр.)
;Перехват int 5
        mov ax,2505h ;Функц.уст.прерывания 5
        lea dx,new_int5 ;Смещ.нов.проц.обр.int 5
        int 21h
        pop ds ;Восстановление DS из стека
        ret
code ends
end simple

```

Программа для отсечения заголовка **EXE** программы **cut512.asm**.

DOS до версии 6.22 включительно имели в своем составе утилиту, превращающую файл программы **.exe** в двоичный файл типа **.bin**, то есть, в файл без заголовка, присущего программам типа **.exe**. Файл этот назывался **exe2bin** (имеется в виду **exe to bin**, то есть, преобразование **.exe** в **.bin**). Однако, в версии **DOS 7** такого файла я не нашел (хотя говорят, что он там есть, возможно под другим названием). Поэтому я написал простую программу, которая отсекает ровно 512 байтов от файла типа **.exe**). Это не полноценная замена программы **exe2bin**, так как **exe2bin** более интеллектуальна – она сама определяет размер заголовка (который всегда кратен 512 байтам, а его размер зависит от размера **.exe** программы). Для нашего случая это несущественно, так как для программы, размером менее 16 Кб (а наши драйверы заведомо всегда меньше), размер заголовка не превышает одного сектора (512 байтов).

Существуют компиляторы, которые позволяют сразу получить файл типа **.bin**) без заголовка.

Компилятор, которым мы пользуемся этой опции не имеет, поэтому приходится идти обходным путем (через файл типа **.exe**). Понятно, что отсечение заголовка (который отвечает за настройку адресов команд в программе, превышающей 64 Кб) не повредит драйверу, так как (как мы уже говорили выше) драйвер не может превышать этого объема.

Итак, программа **cut512.asm** (листинг после описания).

Начало программы стандартное (это программа типа **.com**). Это строки с 4 по 6, и описания не требуют.

Здесь для разнообразия мы определяем главную процедуру (строка 7 – **Start proc near**). Стоит ли напоминать, что ключевое слово **near** (ближняя) можно не писать, так как компилятор по умолчанию делает процедуры ближними (если не указано иное).

Команда **cld** в строке 8 говорит о том, что предполагается использование строковых команд.

Как было указано в пункте 3 подготовки драйвера выше, в командной строке программы **cut512** надо указать файл, у которого надо отсечь заголовок. Результатом работы программы становится файл с тем же именем, но с расширением **.bin** (и, разумеется, без заголовка).

В силу вышесказанного, надо обработать командную строку.

Для этого в строке 10 (**mov si,80h**) в регистр **SI** помещается адрес **80h**, в котором хранится количество символов командной строки. Затем в строке 11 (**xor ch,ch**) обнуляется старший байт **CX** (**CH**), чтобы после строки 12 (**mov cl,[si]**) получить в **CX** 16-разрядный счетчик количества символов командной строки (так как в командной строке счетчик байтовый).

В строке 13 (**lea di,file1**) в **DI** загружается адрес (смещение) буфера для имени исходного файла, а в строке 14 (**lea bx,file2**) в **BX** то же для выходного файла.

Затем в строке 15 (**inc si**) в **SI** осуществляется переход к адресу первого символа командной строки, а в строке 16 (**mov bp,12**) инициализируется счетчик длины имени файла (максимальное значение с расширением – 12, так как точка в длину имени не входит).

В строке 17 (**cmd: lodsb**) начинается цикл переписи имени файла из командной строки в оба буфера. Напоминаю, что **lodsб** строковая команда, которая загружает в аккумулятор (в данном случае, в **AL**, так как в команде в конце стоит буква **b**) один байт из [**SI**], увеличивает **SI** на 1 (так как байт) и уменьшает **CX** на 1.

Далее, в строке 18 (**cmp al,20h**) загруженный символ проверяется на пробел (чтобы его игнорировать). В строке 19 (**jz cmd1**) выполняется условный переход (по нулю, то есть, по совпадению с пробелом) на команду замыкания цикла.

Если переход не выполнен (то есть, символ не был пробелом), в строке 18 (**stosб**) содержимое аккумулятора (**AL**, так как буква **b**) записывается в **ES:[DI]** (в **DI** мы записали ранее смещение первого буфера, а о **ES** позаботилась **DOS** при загрузке **.com** программы). Напоминаю, что команда **stosб**, как и **lodsб** строковая команда. Она загружает из аккумулятора (в данном случае, из **AL**, так как в команде в конце стоит буква **b**) один байт в **ES:[DI]**, увеличивает **DI** на 1 (так как байт) и уменьшает **CX** на 1. Инкремент регистров **SI** и **DI** обусловлен тем, что мы ранее сбросили флаг направления (команда **cld**).

Второй буфер у нас адресует регистр **BX**, поэтому для пересылки того же символа во второй буфер используется команда в строке 21 (**mov [bx],al**). Это команда не строковая, поэтому автоматического инкремента здесь нет, и в строке 22 выполняется обычный инкремент **BX** (**inc bx**).

В следующей строке (строка 23 – **dec bp**) выполняется декремент счетчика символов имени файла (которых не должно быть более 12).

В строке 24 (**jz cmd2**) выполняется условный переход (по нулю) на метку **cmd2**, даже, если символы в командной строке еще не закончились.

На строке 24 (**cmd1: loop cmd**) команда замыкания цикла, которая повторяется, пока в **CX** не нуль, то есть, пока не закончились символы командной строки (если только не произошел ранее выход из цикла по счетчику символов имени файла). В этой строке находится метка **cmd1:**, на которую выполняется переход при обнаружении пробела перед именем файла в командной строке (строка 19).

Далее, со строки 26 начинается цикл анализа имени входного файла. При этом в имени ищется точка, которая обычно ограничивает имя файла, и после которой начинается расширение. Если в имени точка не обнаруживается, то имя файла ограничивается 8 символами (максимальная длина имени без расширения в **DOS**), после чего принудительно вставляется символ точки.

Итак, в строке 26 (**cmd2: mov cx,8**) инициализируется счетчик символов имени файла (без расширения – 8). В строке 27 в регистр **SI** записывается адрес буфера выходного файла (в который ранее было скопировано имя входного файла).

В строке 27 в **SI** загружается смещение второго буфера (**lea si,file2**), а со строки 28 начинается посимвольный анализ имени выходного файла.

Строка 28 (**cmd3: lodsb**) аналогична строке 17. В следующей строке опять делается проверка текущего символа, но уже не на пробел (их не осталось), а на точку (строка 29 – **cmp al,'.'**).

При совпадении (это означает, что имя файла закончилось) в строке 30 (**jz cmd4**) выполняется условный (по нулю) переход на метку **cmd4**, где к имени файла будет приделано расширение **.bin**.

Если это была не точка, управление переходит на строку 31, где замыкается цикл анализа имени (**loop cmd3**). Напоминаю, что максимальное количество повторений цикла определяется исходным значением **CX** (если не был выполнен преждевременный выход из цикла при обнаружении точки), которое в было задано строке 26 (= 8).

Если мы попали на строку 32, значит в имени файла точка обнаружена не была, поэтому здесь на первое место после имени принудительно записывается символ точки (**mov byte ptr[si],'**).

Затем адрес в **SI** инкрементируется (**inc si** для перехода к первому символу расширения).

Со строки 34 в имя файла тупо вбивается расширение **.bin**:

строка 34 – **cmd4: mov byte ptr[si],'b'**,

строка 35 – **mov byte ptr[si+1],'i'**,

строка 36 – **mov byte ptr[si+2], 'n'**.

С именем выходного файла возня закончилась. Теперь пора усекать в исходном файле первые 512 байтов. Для этого, в первую очередь, необходимо этот файл открыть:

- строка 38 (**mov ah,3dh**) функция открытия файла,
- строка 39 (**mov al,2**) указание режима открытия *чтение и запись),
- строка 40 (**lea dx,file1**) запись в **DX** смещения спецификации (имени) файла (о содержимом **DS** позаботилась **DOS** при запуске программы),
- строка 41 (**int 21h**) вызов системного прерывания.

Как всегда после файловой функции, проверяем, не было ли ошибки (не взведен ли флаг переноса) – строка 42 (**jnc m1**). Если ошибки не было (переход по сброшенному флагу переноса), переходим на дальнейшую обработку.

Если ошибка все же была, в строке 43 (**jmp err1**), куда мы попадаем, если не было перехода в строке 42, выполняется переход на вывод сообщения о файловой ошибке с последующим выходом из программы.

На строку 44 (**m1: mov handle,ax**) мы попадаем, если ошибки не было. Здесь мы сохраняем дескриптор открытого файла (который функция **3dh** возвращает в регистре **AX**) в переменной **handle**.

Теперь нам надо определить длину файла. Для этого мы сначала устанавливаем указатель файла на его конец:

- строка 46 (**mov ah,42h**) – функция установки указателя,
- строка 47 (**mov al,2**) – установка относительно конца файла,
- строка 48 (**mov bx,handle**) – дескриптор файла,
- строка 49 (**xor cx,cx**) – старшая половина смещения (0),
- строка 50 (**xor dx,dx**) – младшая половина смещения (0),
- строка 51 (**int 21h**) – системное прерывание.

Далее, как при любой файловой операции переход по возможной файловой ошибке (строка 52 – **jc err1**). В этой программе все файловые ошибки обрабатываются однотипно – сообщение об ошибке и выход из программы.

Если ошибки не было, мы попадаем на строку 53 (**mov len,ax**), где сохраняем возвращенную в **AX** длину файла в переменной **len**.

Теперь надо установить указатель в файле на начало:

- строка 55 (**mov ah,42h**) – функция установки указателя,
- строка 56 (**xor al,al**) – знаковое смещение от начала файла,
- строка 57 (**mov bx,handle**) – дескриптор файла,
- строка 58 (**xor cx,cx**) – старшая половина смещения (0),
- строка 59 (**xor dx,dx**) – младшая половина смещения (0),
- строка 60 (**int 21h**) – системное прерывание,
- строка 61 (**jc err1**) – стандартный переход, если ошибка.

Если ошибка не случилась, попадаем на строку 63, начиная с которой выполняется чтение файла в буфер:

- строка 63 (**mov ah,3fh**) – функция чтения из файла,
- строка 64 (**mov bx,handle**) – дескриптор файла,
- строка 65 (**mov cx,len**) – длина считываемого текста
- строка 66 (**lea dx,text**) – адрес буфера записи,
- строка 67 (**int 21h**) – системное прерывание,
- строка 68 (**jc err1**) – стандартный переход, если ошибка.

Если ошибки не было, попадаем в строку 70, с которой начинается закрытие исходного файла:

- строка 70 (**mov ah,3eh**) – функция закрытия файла,
- строка 71 (**mov bx,handle**) – дескриптор файла
- строка 72 (**int 21h**) – системное прерывание,
- строка 73 (**jc err1**) – переход, если ошибка.

Если ошибки не было, попадаем в строку 75, с которой начинается создание нового (выходного) файла:

- строка 75 (**mov ah,3ch**) – функция создания файла,
- строка 76 (**mov cx,0**) – без атрибутов,
- строка 77 (**lea dx,file2**) – адрес спецификации файла,
- строка 78 (**int 21h**) – системное прерывание,
- строка 79 (**jc err1**) – переход, если ошибка.

Если ошибки не было, попадаем в строку 80 (**mov handle,ax**), в которой сохраняем дескриптор созданного файла (который функция **3ch** возвращает в регистре **AX**) в переменной **handle**. Обращаем внимание на то, что файл создается с именем, которое было сформировано в буфере имени выходного файла.

После этого в строке 82 начинается процесс записи данных в выходной файл:

- строка 82 (**mov ah,40h**) – функция записи в файл,
- строка 83 (**mov bx,handle**) – дескриптор файла,
- строка 84 (**mov cx,len**) – длина исходного файла,
- строка 85 (**sub cx,512**) – длина записываемого файла (на 512 байтов меньше, чем исходного, так как заголовок будет удален),
- строка 86 (**lea dx,text+512**) – адрес записываемого текста в исходном файле, смещенный на 512 байтов от начала,
- строка 87 (**int 21h**) – системное прерывание,
- строка 88 (**jc err1**) – переход, если ошибка.

Если запись прошла успешно (без ошибки), то мы попадаем на строку 90, где начинается процесс закрытия выходного файла (иногда это забывают сделать, что плохо):

- строка 90 (**mov ah,3eh**) – функция закрытия файла,
- строка 91 (**mov bx,handle**) – дескриптор файла,
- строка 92 (**int 21h**) – системное прерывание,
- строка 93 (**jc err1**) – переход, если ошибка,
- строка 94 (**int 20h**) – на этом все.

Осталась реакция на файловую ошибку и переменные с сообщениями.

Реакция на файловую ошибку:

- строка 96 (**err1: mov ah,9**) – стандартная функция вывода строки на экран,
- строка 97 (**lea dx,meserr**) – адрес сообщения об ошибке,
- строка 98 (**int 21h**) – системное прерывание,
- строка 99 (**int 20h**) – выход из программы, так как после файловой ошибки больше делать нечего.

Переменные:

- строка 100 (**file1 db 13 dup(0)**) – буфер для имени исходного файла (максимально 13 символов – 8 имя, 3 расширение и точка),
- строка 101 (**file2 db 13 dup(0)**) – буфер для имени выходного файла,
- строка 102 (**meserr db 'File error\$'**) – текст сообщения о файловой ошибке,
- строка 103 (**handle dw ?**) – переменная для дескриптора файла (только резервирует память),
- строка 104 (**len dw ?**) – переменная для длины файла (тоже только резервирует память),
- строка 105 (**text db 4000h dup(0)**) буфер для содержимого исходного файла (отводится только 16 килобайт; можно больше, но ни к чему).

Строка 106 (**Start endp**) – замыкание главной процедуры.

Последние две строки – стандартное завершение программы.

Листинг программы **cut512.asm**.

```
; Усечение заголовка EXE программы на 512 байтов
; Максимальный размер усекаемой программы около 16 КВ,
; что определяется размером буфера text в конце программы
Assume CS: Code, DS: Code
Code    SEGMENT
        org 100h
Start   proc    near
        cld
; Обработка командной строки
        mov     si,80h ;Нач.ком.строки
        xor     ch,ch  ;Обнуление CH
        mov     cl,[si];В CL кол-во симв.ком.стр.
        lea     di,file1;Буфер для имени исх.файла
        lea     bx,file2;Буфер для имени вых.файла
        inc     si ;1 симв. командной строки
        mov     bp,12 ; Макс. длина имени файла
cmd:    lodsb   ;Симв.ком.стр.из [SI]в AL
        cmp     al,20h;Проверка на пробел
        jz      cmd1 ;Пробел (пропустить)
        stosb  ;Симв.из AL в [DI] (исх.файл)
        mov     [bx],al;Его же в [BX] (вых.файл)
        inc     bx
```

```

        dec     bp ;Сч.симв.имени файла с расш.
        jz     cmd2 ;Длина имени = 12
cmd1:   loop    cmd ;К след.симв.ком.строки
cmd2:   mov     cx,8 ;Сч.симв.имени файла без расш.
        lea    si,file2 ;Буфер для имени вых.файла
cmd3:   lodsb   ;Посим.чтение имени вых.файла
        cmp    al,'.' ;Проверка на точку
        jz     cmd4 ;Точка найдена
        loop   cmd3 ;Перех.если не точка
        mov    byte ptr[si], '.' ;Зап.точки,если не найдена
        inc    si ;К след.байту
cmd4:   mov    byte ptr[si],'b' ;Новое расширение
        mov    byte ptr[si+1],'i'
        mov    byte ptr[si+2],'n'
; Открытие файла
        mov    ah,3dh ; Функция открытия файла
        mov    al,2 ; Запись и чтение
        lea    dx,file1; Адрес спецификации файла
        int    21h ; Функция DOS
        jnc    m1 ;Если нет ошибки
        jmp    err1 ; Переход, если ошибка
m1:     mov    handle,ax ;Сохр.дескриптора файла
; Определение длины файла
        mov    ah,42h ; Функция установки указателя
        mov    al,2 ; Знаковое смещение от конца файла
        mov    bx,handle; Дескриптор файла
        xor    cx,cx ;Ст.половина смещения (0)
        xor    dx,dx ;Мл.половина смещения (0)
        int    21h ; Функция DOS
        jc    err1 ; Переход, если ошибка
        mov    len,ax ;Сохр.дл.файла (одно слово)
; Установка указателя на начало
        mov    ah,42h ;Функция установки указателя
        xor    al,al ;Знаковое смещение от начала файла
        mov    bx,handle; Дескриптор файла
        xor    cx,cx ;Ст.половина смещения (0)
        xor    dx,dx ;Мл.половина смещения (0)
        int    21h ;Функция DOS
        jc    err1 ; Переход, если ошибка
; Чтение файла
        mov    ah,3fh ;Функция чтения из файла
        mov    bx,handle ;Дескриптор файла
        mov    cx,len ;Длина считываемого текста
        lea    dx,text ;Адрес буфера записи
        int    21h
        jc    err1 ;Переход, если ошибка
; Заккрытие файла
        mov    ah,3eh ; Функция закрытия файла
        mov    bx,handle ; Дескриптор файла
        int    21h
        jc    err1 ;Переход, если ошибка
; Создание нового файла
        mov    ah,3ch ;Функция создания файла

```

```

        mov     cx,0      ;Без атрибутов
        lea    dx,file2; Адрес спецификации файла
        int    21h      ; Функция DOS
        jc     err1     ; Переход, если ошибка
        mov    handle,ax ; Сохранение дескриптора файла
; Запись в файл
        mov    ah,40h    ;Функция записи в файл
        mov    bx,handle ;Дескриптор файла
        mov    cx,len    ;Длина считываемого текста
        sub    cx,512    ;- длина заголовка (512 байт)
        lea    dx,text+512;Адр.запис.текста
        int    21h      ; (с пропуском загол.)
        jc     err1     ;Переход, если ошибка
; Закрытие файла
        mov    ah,3eh    ; Функция закрытия файла
        mov    bx,handle ; Дескриптор файла
        int    21h
        jc     err1     ;Переход, если ошибка
        int    20h      ;Все
; Вывод сообщения при возникновении ошибки
err1:   mov    ah,9      ; Функция вывода строки на экран
        lea    dx,meserr ; Адрес сообщения об ошибке
        int    21h
        int    20h
file1   db     13 dup(0)
file2   db     13 dup(0)
meserr  db     'File error$'
handle  dw     ?        ; Место для дескриптора файла
len     dw     ?        ; Длина
text    db     4000h dup(0)
Start   endp
Code    ENDS
        END      Start

```

Резидентная программа `tsr1.asm`, которая делает то же, что и драйвер, но выводит символ `$`. Эта программа здесь приводится для того, чтобы наглядно убедиться, что драйвер занимает существенно меньше места в памяти, чем такая же по функциям резидентная программа.

Данная резидентная программа достаточно проста, и, по моему, не требует пояснений. Если вопросы, все же, возникнут, пишите.

Листинг программы **tsr.asm**.

```

; TSR, выводящая на экран символ по PrtScr
Assume CS: Code, DS: Code
Code    SEGMENT
        org 100h
sym     equ '$'
resprog proc far
        jmp  init
; Новый обработчик прерывания 05h
new_int5 proc far;Дальняя, так как вызывается извне
        push ax ;Сохранение AX в стеке (он портится)
        mov  ah,0eh ;Вывод в режиме телетайпа

```

```

    mov  al,sym;7;cs:sym ;CS обязательно
    int  10h
    pop  ax ;Восстановление AX из стека
    iret
new_int5 endp ;Конец новой процедуры обр.прер.5
resprog endp ;Конец проц.резидентного кода
ressize equ  $-resprog;Размер в байтах резидентной части
init: mov  ax,2505h ;Функция 25h, вектор 5
      lea  dx,new_int5;Адрес новой проц.int 5
      int  21h ;Запись нового вектора lch
;Завершение программы инициализации с оставлением
;резидентной части в памяти
      mov  dx,(ressize+10fh);Размер резидента в байтах
      int  27h;Завершение с оставлением резид.кода
Code  ENDS ;Конец сегмента кода
      END resprog ;Точка входа

```

Для получения и установки резидентной программы необходимо:

1. Скомпилировать ее:

tasm tsr.asm – получится **tsr.obj**

2. Скомпоновать:

tlink tsr.obj /t – получится **tsr.com**

3. Запустить **tsr.com**

4. Попробовать понажимать клавишу **PrtScr**, и убедиться, что на экран выводится символ **\$**.

Запустить программу **mi.com**, которая покажет расположение драйверов и программ в памяти. Убедиться, что драйвер занимает в памяти 80 байтов, а резидентная программа, делающая то же, что и драйвер, – 368 байтов. Попробуйте объяснить этот факт.

В папке находится скриншот экрана виртуальной машины **DOS** работы программы **mi.com**.