МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

А.В. РОЩИН

ФУНКЦИОНАЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВСТРОЕННЫХ СИСТЕМ

РЕЗИДЕНТНЫЕ ПРОГРАММЫ, ДРАЙВЕРЫ, РАБОТА С 32-РАЗРЯДНЫМИ ПРОЦЕССОРАМИ

УЧЕБНОЕ ПОСОБИЕ

Москва — 2017

УДК 681.3
ББК 22.18
А __

Рощин А.В. Функциональное программное обеспечение встроенных систем. Резидентные программы и драйверы. Работа с 32-разрядными процессорами. [Электронный ресурс]: ученое пособие — М., Московский технологический университет (МИРЭА), 2017 — 1 электрон. опт. диск (CD-ROM)

Рекомендовано Ученым Советом Физико-технологического института в качестве учебного пособия для специальности 09.03.01.

Настоящее учебное пособие предназначено для подготовки студентов различных вычислительных специальностей, изучающих работу встроенных систем, работающих с 16 и 32-разрядными микропроцессорами семейства x86 в реальном режиме. Для специальности 09.03.01 это пособие может использоваться в курсах "функциональное программное обеспечение встроенных систем", "Проектирование микропроцессорных систем", "Организация ввода-вывода".
В пособии достаточно подробно описаны особенности работы с 16 и 32-разрядными микропроцессорами семейства x86 с точки зрения программиста, рассмотрены расширенные возможности адресации таких процессоров – использование 32-разрядных операндов и режима линейной адресации в реальном режиме. Рассмотрены листинги соответствующих программ.

Учебное пособие предназначено для студентов всех форм обучения направления подготовки 09.03.01 «Информатика и вычислительная техника» и 09.04.01 «Информатика и вычислительная техника».

Учебное пособие издается в авторской редакции.

Автор: Рощин Алексей Васильевич

Рецензенты:

Туманов Михаил Петрович, профессор, к.т.н. Департамента электронной инженерии МИЭМ НИУ ВШЭ им. А.Н.Тихонова

Рязанова Наталья Юрьевна, доцент, к.т.н. кафедры ИУ7 МГТУ им. Баумана

Введение

Данная часть учебного пособия посвящена созданию и использованию разидентных кодов в однозадачной операционной системе – резидентных программ и драйверов.

Рассмотрена структура резидентной программы, порядок ее создания и установки резидентом. Рассмотрены различные варианты оставления резидентного кода в пямяти.

Показаны возможности защиты памяти от повторной загрузки резиджентной программы, а также аозможности ее выгрузки.

Рассмотрены структура и особенности драйверов под DOS. Рассмотрены команды драйвера, процедура его загрузки операционной системой, а также порядок обращения к нему.

Показан процесс создания жрайвера,

Рассмотрены листинги резидентных программ и драйверов.

Рассмотрены проблемы, связанные с эжксплуатацией 32-разрядных процессоров, повобы адпесации в защищенном режиме.

Рассмотрены также методы использования рижема линейной адресации памяти большого объема из реального режима.

Тема 1 Резидентные программы в MS-DOS

1.1 Специфика резидентных программ

Резидентная программа – это программа, постоянно находящаяся в оперативной памяти ЭВМ. Иначе такие программы называют TSR-программами (Terminate and Stay Resident). В качестве резидентных программ часто выполняют различные обработчики клавиатуры (в том числе русификаторы) калькуляторы, всевозможные справочники и т.д.

Резидентная программа может быть как типа .COM, так и .EXE, однако, учитывая постоянный дефицит основной памяти, такие программы чаще выполняют типа .COM.

Для того чтобы использовать уже находящуюся в памяти программу, ей необходимо передать управление. Специфика передачи управления резидентной программе заключается в том, что вызывающая и вызываемая (резидентная) программы загружаются и запускаются независимо друг от друга, поэтому необходимы специальные меры для сообщения вызывающей программе адреса резидентной программы.

Передать управление резидентной программе можно тремя способами:

Вызвать ее командой CALL как обычную процедуру (под программу). Однако для этого необходимо после загрузки резидентной программы узнать ее расположение в памяти с помощью какой-либо служебной программы, например mi (memory information);

- Использовать какое-либо аппаратное прерывание (например, прерывание от таймера) для периодической пере дачи управления резидентной программе;
- Использовать программное прерывание. Для этого резидентная программа должна соответствующим образом установить вектор программного прерывания, который будет использован для ее вызова. Для пользователя в MS-DOS зарезервированы векторы 60h – 66h, а также F1h
- FFh. В этом случае резидентная программа должна завершаться командой возврата из прерывания IRET.

Адрес резидентной программы можно передать прикладной программе также в области данных BIOS, предназначенной для связи программ (40h:F0h – 40h:FFh). В этой же области прикладная программа может передавать адреса массивов данных, которые должны быть переданы резидентной программе, а также получать адреса массивов данных, возвращаемых резидентной программой.

Резидентная программа после загрузки ее в память фактически становится частью операционной системы, поэтому к ней относится и такое свойство MS-DOS, как нереентерабельность (т.е. она не обладает свойством повторной входимости). Это связано с тем, что MS-DOS разрабатывалась, как однозадачная операционная система, и в ней используются внутренние рабочие области, которые могут быть испорчены при попытке параллельного выполнения нескольких процессов. Практическим следствием этого свойства является тот факт, что резидентная программа не может использовать большую часть функций MS-DOS и BIOS. Эти функции может использовать инициализирующая часть резидентной программы, так как в момент загрузки резидентная программа еще не является частью операционной системы.

После первой загрузки резидентной программы в память должны пресекаться все последующие подобные попытки, так как повторная загрузка может привести к более или менее крупным неприятностям. Следить за этим должна сама резидентная программа.

1.2 Структура резидентной программы

Резидентная программа состоит, как правило, из двух частей – резидентной секции (которая обычно располагается вначале) и инициализирующей (которая обычно расположена в конце).

При первом запуске резидентная программа загружается в память целиком, и управление передается инициализирующей секции, которая проверяет, не находится ли уже резидентная секция этой программы в памяти. Если такая программа уже присутствует, выводится соответствующее сообщение и дальнейшее выполнение программы прекращается без последствий. Если такой программы нет в памяти, выполняются следующие действия:

настраиваются все необходимые векторы прерываний (при этом могут устанавливаться новые векторы и модифицироваться старые);

если необходимо, заполняются все области указателей адресов передачи управления и данных;

программа настраивается на конкретные условия работы (возможно заданные в командной строке при запуске резидентной программы);

завершается выполнение инициализирующей части при помощи функции 31h прерывания DOS int 21h или при помощи прерывания DOS int 27h. При этом резидентная секция программы, размер которой инициализирующая секция передает DOS, остается в памяти.

Следует отметить, что важнейшей функцией инициализирующей секции резидентной программы является указание DOS размера оставляемой

резидентной секции программы. Если для завершения инициализирующей секции используется прерывание DOS int 27h, в регистре dx указывается размер резидентной секции в байтах. При этом следует иметь в виду, что в этот размер входят также 100h байтов префикса программного сегмента

PSP. Ясно, что с помощью этого прерывания DOS нельзя оставить в памяти программу, больше 64 килобайт. Если для завершения инициализирующей секции используется функция 31h прерывания DOS int 21h, в регистре dx указывается размер резидентной секции (с учетом PSP) в параграфах (1 параграф = 16 байтам). Для определения размера резидентной секции в параграфах вычисляется выражение (size + 100h + 0Fh)/16 где: size – размер резидентной секции в байтах. Дополнительное слагаемое 0Fh (десятичное 15) в выражении необходимо для того, чтобы отводимое количество параграфов было округлено в большую сторону. В противном случае будет отсечен конец программы, меньший параграфа.

Ранее уже было сказано о том, что инициализирующая секция располагается в конце программы. Такое расположение приводит к тому, что после завершения инициализирующей секции занимаемая ею память освободится, так как она не входит в указанный размер и расположена после резидентной секции.

Листинг структуры резидентной программы типа .COM

```
Code SEGMENT
assume cs:Code, ds:Code
org     100h
------------------------------------------------------------
resprog         proc    far
        jmp     init      ; Переход на секцию инициализации
 ; Данные и переменные резидентной секции
 .....................
entry:                          ; Текст резидентной секции
 .....................
resprog endp
------------------------------------------------------------
size equ $-resprog              ; Размер резидентной секции в байтах
------------------------------------------------------------
init            proc    near    ; Инициализирующая секция
 ; Текст инициализирующей секции
 ; Вычисление (size + 10Fh)/16 -> DX
        mov     ax,3100h        ; Функция 31h
        int     21h
init    endp
------------------------------------------------------------
Code  ends
```

```
END   resprog
```

Пояснения к листингу структуры резидентной программы:

1. Предполагается, что прописные и строчные буквы транслятором не различаются (по умолчанию так и есть).
2. Процедура resprog объявлена как дальняя, так как в ней находится текст резидентной секции, управление которой может передаваться только с помощью дальнего перехода или вызова.
3. В тексте резидентной секции должна быть предусмотрена команда возврата в вызывающую прикладную программу. Это может быть команда IRET, если резидентная программа вызывается при помощи программного прерывания int, это может быть просто RET, если резидентная программа вызывается, как подпрограмма командой CALL, это может быть и что-нибудь более экзотическое – фантазии программистов нет предела.
4. Процедура init объявлена как ближняя, так как вызывающая процедура находится в том же сегменте.

1.3 Обращение к резидентной программе

Для обращения к резидентной программе, как уже было сказано, можно использовать область данных BIOS, предназначенную для связи между процессами (40h:F0h – 40h:FFh). Эта область не используется операционной системой, поэтому использование ее для вызова резидентной программы вроде бы не предвещает ничего неожиданного. Так оно и есть, если разработчик одной (или не одной) резидентной программы, уже находящейся в памяти, не использовал ту же область для тех же целей. (Кстати, это относится и ко всем другим способам обращения к резидентной программе.) Мы не будем рассматривать такую возможность, хотя и в этом случае есть простор для творчества.

Итак, имеется область размером 16 байтов, которая может быть использована по желанию. Так как полный адрес, необходимый для дальнего вызова или перехода требует четырех байтов, в этой области можно разместить 4 таких адреса. Это может быть адрес входа в резидентную секцию и 3 адреса, указывающих на таблицы данных, расположенных где-то еще в памяти. Можно использовать эту область так – адрес точки входа в резидентную секцию (4 байта) и 12 байтов непосредственных данных. Возможны различные промежуточные варианты.

Рассмотрим вариант с двумя адресами – адресом точки входа в резидентную секцию и адресом таблицы параметров (tabl_param) в сегменте

данных прикладной программы, которая должна быть передана резидентной программе. Для обеспечения взаимодействия инициализирующая секция резидентной программы записывает в слово 40h:F0h – смещение точки входа в резидентную секцию (например, offset entry), в слово 40h:F2h – содержимое сегментного регистра CS.

Прикладная программа для вызова резидентной программы должна, например, настроить сегмент расширения на начало области данных BIOS (ES = 40h) и выполнить команду дальнего вызова call dword ptr es:0F0h

Конечно, резидентная программа должна быть объявлена дальней процедурой и завершаться соответствующей командой дальнего возврата RET (впрочем, ее можно явно сделать дальней – RETF).

Для передачи резидентной программе адреса таблицы параметров прикладная программа должна записать в слово 40h:F4h – смещение начала таблицы параметров в сегменте данных прикладной программы (offset tabl_param), а в слово 40h:F6h – текущее содержимое сегментного регистра DS.

Резидентная программа для получения этих данных должна поместить в какой-либо регистр, например SI, смещение начала таблицы из 40h:F4h, а в сегментный регистр, например в DS, сегментный адрес из 40h:F6h, после чего резидентная программа получает доступ к самим данным. Возможная последовательность команд в резидентной программе может быть такой

```
mov   es,40h        ; ES на начало области данных BIOS
mov   bx,0F4h
mov   si,es:[bx]    ; SI = offset tabl_param
mov   bx,0F6h
mov   ax,es:[bx]    ; AX – сегм. адрес tabl_param
mov   ds,ax
mov   ax,[si]       ; Первое слово данных
mov   bx,[si+2]     ; Второе слово данных и т.д.
```

Не следует забывать сохранять в резидентной программе все используемые ею регистры и восстанавливать их перед выходом из программы. При этом следует осторожно пользоваться стеком для сохранения регистров, так как системный стек не очень велик, а заводить собственный стек в резидентной программе не всегда целесообразно. Можно сохранять регистры в специально отведенных для этого рабочих переменных.

Более удобно использовать для вызова резидентной программы один из свободных векторов прерывания (векторы 60h – 66h, а также F1h – FFh). Инициализирующая секция резидентной программы должна поместить свой адрес в один из свободных векторов, например, F1h:

```
mov  ax,0
mov  es,ax
mov  es:0F1h*4,offset entry      ; Адрес вектора F1h
mov  es:0F1h*4+2,cs
```

В результате этой последовательности команд в векторе F1h окажется адрес точки входа в резидентную программу. Для вызова резидентной программы в этом случае достаточно использовать команду int 0F1h. В этом случае резидентная программа, как и все программы обработки прерывания должна завершаться командой возврата из прерывания IRET. Адреса таблиц параметров можно передавать прежним способом, а можно и через другие свободные векторы прерываний.

1.4 Защита от повторной загрузки

Для защиты от повторной загрузки резидентной программы в память инициализирующая секция должна предпринять некоторые действия по обнаружению собственной резидентной секции в памяти, а резидентная секция должна соответствующим образом ответить на эти действия. Для осуществления этих действий можно использовать мультиплексное прерывание DOS int 2Fh.

Функции C0h – FFh этого прерывания зарезервированы для пользователя. В DOS принято, что прерывание 2Fh возвращает в регистре AL следующие состояния резидентной программы:

0 – программа не установлена, но ее можно установить;

1 – программа не установлена, и ее нельзя установить;

FFh – программа установлена.

При ошибке должен быть установлен флаг переноса CF, а в регистре AX следует вернуть код ошибки. Для того чтобы резидентная секция программы реагировала на прерывание 2Fh, в нее следует включить обработчик соответствующих функций этого прерывания. Для нормальной работы этого обработчика инициализирующая секция должна установить новый вектор прерывания 2Fh, сохранив при этом старый вектор во внутренней переменной. Новый обработчик прерывания 2Fh должен выполнить все, что ему положено, а после этого вызвать старый обработчик этого прерывания. В приведенном ниже листинге резидентной программы использован именно этот способ защиты от повторной загрузки.

Другим способом защиты от повторной загрузки является использование специального кода для индикации наличия резидентной программы в памяти. Специальный идентифицирующий код помещается в заранее определенное место в памяти или в заранее определенное место в резидентной секции программы. Если код помещается в определенное место

в памяти (например, на месте вектора прерывания 60h), при инициализации проверяется наличие этого кода в этом месте. Если код в наличии, загрузка программы не производится.

Если идентифицирующий код (сигнатура) помещается в определенном месте резидентной секции, инициализирующая секция проверяет наличие этого кода по адресу точки входа в резидентную программу (она знает, как определить адрес точки входа) и по положению этого кода относительно точки входа (это она тоже знает). Обнаружение кода влечет за собой отказ от загрузки программы.

1.5 Использование командной строки

При запуске программы DOS формирует префикс программного сегмента (PSP), который загружается в память перед программой. Сразу после загрузки DS:0000 и ES:0000 указывают на начало PSP этой программы. Информация, содержащаяся в PSP позволяет выделить имена файлов и всевозможные ключи из командной строки, узнать объем доступной памяти, определить окружение и т. д.

Структура префикса программного сегмента приведена ниже. Для использования командной строки ее следует считать из PSP, учитывая, что сразу после запуска программы .EXE сегментные регистры DS и ES настроены на начало PSP. В случае программы .COM на начало PSP настроены все сегментные регистры (CS, DS, ES, SS).

Ниже приведен фрагмент программы выполняющей анализ командной строки.

Смещ.	Длина	Содержимое	
+0	2	int 20h	Выход в DOS
+2	2	Mem Top	Вершина доступной памяти в параграфах
+4	1		(Резерв)
+5	5	Call смещ. сегмент	Вызов диспетчера функций DOS
+0ah	4	смещ. сегмент	Адрес завершения (см. int 22h)
+0eh	4	смещ. сегмент	Адрес обр. Ctrl-Break (см. int 23h)
+12h	4	смещ. сегмент	Обраб. критич. ошибок (см. int 24h)
+16h	16h	Резервная область DOS	
+2ch	2	Env Seg	Сегментный адрес окружения
+2eh	2eh	Резервная область DOS	

+5ch	10h	FCB1	FCB первого параметра команды
+6ch	14h	FCB2	FCB второго параметра команды
+80h	1	len	Длина области UPA (с адр. 81h) или DTA
+81h	7fh	Неформ. обл. параметров	Символы ком. строки DOS

```
mov   bx,80h
mov   cx,[bx]       ; Кол. символов командной строки
inc   bx            ; Начало командной строки
cmd:  mov   al,[bx]       ; Первый символ командной строки
      Здесь производится анализ командной строки
inc   bx
loop  cmd
```

Во втором листинге резидентной программы выполняется анализ командной строки для распознавания ключа выгрузки /u.

1.6 Листинги резидентных программ

Написать резидентную программу, которая перехватывает прерывание int 5 (Print Screen) и вместо распечатки экрана на каждое нажатие клавиши PrtSc изменяет цвет рамки экрана. Рамка должна принимать циклически один из 16 цветов. Программа не должна позволять загрузить себя повторно. При попытке повторной загрузки программа должна выводить предупреждающее сообщение.

Указания:

- Передавать управление старому обработчику прерывания не надо.
- Для окрашивания рамки экрана следует использовать подфункцию 01h функции 10h прерывания 10h (ax = 1001h, bx = цвет).
- В начале программы следует не забыть записать в DS значение CS.
- Для проверки наличия резидентной программы в памяти использовать функцию FFh прерывания 2Fh.

```
Assume CS: Code, DS: Code
Code  SEGMENT
      org    100h
resprog proc far
mov   ax,cs
mov   ds,ax
jmp   init
color                db     0
old_int2Fh_off       dw     ?
old_int2Fh_seg       dw     ?
```

```
msg                 db      'Драйвер уже установлен$'
; Новый обработчик прерывания 2Fh
new_int2Fh  proc    far
        cmp     ax,0ff00h
        jz      installed
        jmp     dword ptr cs:old_int2Fh_off
installed:      mov     ax,0ffh
        iret
new_int2Fh  endp
; Новый обработчик прерывания 5
new_int5    proc    far
        mov     bh,color
        inc     color
        mov     ax,1001h
        int     10h
        iret
new_int5    endp
resprog     endp
ressize     equ     $-resprog       ; Размер в байтах резидентной части
init    proc    near
; Проверка раличия резидентной программы в памяти
mov     ax,0ff00h
int     2fh
cmp     ax,0ffh
jnz     first_start     ; Не установлена
lea     dx,msg          ; Вывод сообщения о том,
mov     ah,9            ; что драйвер уже загружен
int     21h
ret
first_start:    mov     ax,2505h        ; Функция 25h, вектор 5
        lea     dx,new_int5
        int     21h             ; Запись нового вектора 5
        mov     ax,352fh        ; Сохранение старого вектора прерывания 2Fh
        int     21h
        mov     cs:old_int2fh_off,bx
        mov     cs:old_int2fh_seg,es
        lea     dx,new_int2Fh ; Запись нового вектора прерывания 2Fh
        mov     ax,252fh
        int     21h
; Завершение программы с оставлением резидентной части в памяти
        mov     dx,(ressize+10fh)/16
        mov     ax,3100h
        int     21h
init    endp
Code  ENDS
END   resprog
```

Написать резидентную программу, записывающую содержимое экрана в символьном режиме в файл. Программа должна анализировать флаг активности DOS и не должна допускать повторной загрузки в память. По ключу /u программа должна выгружаться из памяти с освобождением занимаемого ей места. Замечание: приведенная ниже программа нормально работает лишь под DOS до версии 5.0, так как в более поздних версиях иначе происходит работа с клавиатурой.

```
Code SEGMENT
Assume CS: Code, DS: Code
org     100h
resprog proc far
mov     ax,cs
mov     ds,ax  ; DS = CS
jmp     init     ; Переход на инициализирующую секцию
num                     dw      0       ; Количество сброшенных экранов
old_int8_off            dw      ?       ; Адрес старого обработчика
old_int8_seg            dw      ?       ; прерывания таймера 8h
old_int5_off            dw      ?       ; Адрес старого обработчика
old_int5_seg            dw      ?       ; прерывания 5h
old_int2F_off           dw      ?       ; Адрес старого обработчика
old_int2F_seg           dw      ?       ; мультиплексного прерывания 2Fh
adr_psp                 dw      ?       ; Адрес PSP
vbuf                    dw      0b000h          ; Сегментный адрес видеобуфера
handle                  dw      ?       ; Дескриптор файла
buf                     db      2050 dup(0)  ; Буфер для данных экрана
mes                     db      'Disk error$'
filename                db      'filescr&.txt',0 ; Спецификация вых. файла
iniflag         db      0       ; Флаг запроса на вывод экрана в файл
outflag                 db      0       ; Флаг начала вывода в файл
_crit                   dd      ?       ; Указатель на флаг активности DOS
; Новый обработчик прерывания 2Fh
new_int2F       proc    far
cmp     ax,0ff00h
jz      installed
jmp     dword ptr cs:old_int2F_off ; Переход на старый обработчик 2Fh
installed:      mov     ax,0ffh         ; "Программа в памяти"
                iret
new_int2F       endp
; Новый обработчик прерывания 1ch
new_int8        proc    far
        push    ax
        push    bx
        push    cx
        push    dx
```

```
        push   si
        push   di
        push   ds
        push   es
        mov    ax,cs
        mov    ds,ax
        cmp    iniflag,0
        jz     exit8          ; Нет запроса
        test   outflag,0ffh
        jnz    exit8          ; Файл уже выводится
        jnz    exit8          ; DOS занята
        les    bx,_crit
        test   byte ptr es:[bx],0ffh
        jnz    exit8          ; DOS занята
; iniflag=1, outflag=0, crit=0
        mov    outflag,0ffh
        call   writef         ; Вывод буфера в файл
exit9:          pop    es
        pop    ds
        pop    di
        pop    si
        pop    dx
        pop    cx
        pop    bx
        pop    ax
        iret
new_int8      endp
; Новый обработчик прерывания 5
new_int5      proc   far
        mov    cs:iniflag,0ffh
        iret
new_int5      endp
; Запись видеобуфера в файл
writef  proc   near
mov    ax,cs
mov    ds,ax
mov    ax,0e07h
        int    10h
mov    ax,vbuf        ; Начало видеобуфера (сегмент)
mov    es,ax
mov    si,0
lea    di,buf
mov    dx,25          ; Число строк
cld
trans1:         mov    cx,80  ; Число символов в строке
trans:          mov    al,es:[si]
```

```
        mov   [di],al
        inc   si
        inc   si
        inc   di
        loop  trans
        mov   byte ptr [di],0dh
        inc   di
        mov   byte ptr [di],0ah
        inc   di
        dec   dx
        jnz   trans1
; Создание файла
        test  word ptr num,0ffffh  ; Сброшено экранов 0 ?
        jnz   sdwig          ; Переход на смещение указателя
        mov   word ptr num,2050
        mov   ah,3ch         ; Функция создания файла
        mov   cx,0           ; Без атрибутов
        lea   dx,filename    ; Адрес спецификации файла
        int   21h
        jc    noform
        mov   handle,ax      ; Сохранение дескриптора файла
        jmp   write
sdwig:  mov   ax,3d01h       ; Открытие файла с записью
        lea   dx,filename
        int   21h
        jc    noform
        mov   handle,ax
        mov   ax,4200h       ; Установка указателя файла
        mov   bx,handle
        mov   cx,0
        mov   dx,num
        add   word ptr num,2050
        int   21h
; Запись файла
write:  mov   ah,40h         ; Функция записи в файл
        mov   bx,handle      ; Дескриптор файла
        mov   cx,2050        ; Длина записываемого массива
        lea   dx,buf         ; Адрес записываемого массива
        int   21h
        jc    noform
;Закрытие файла
        mov   ah,3eh         ; Функция закрытия файла
        mov   bx,handle      ; Дескриптор файла
        int   21h
        jmp   prend
noform: mov   ah,9
```

```
        mov   dx,offset mes
        int   21h
prend:  mov   outflag,0
        mov   iniflag,0
        ret
writef  endp
resprog endp
ressize equ   $-resprog     ; Размер в байтах резидентной части
init    proc  near
; Проверка ключа /u
mov   bx,80h
mov   cx,[bx]         ; Кол. символов в командной строке
inc   bx              ; Начало командной строки
cmd:  mov   al,[bx]
      cmp   al,20h
jz    cmd1                ; Пробел
cmp   al,'/'
jnz   cmd2                ; Не ключ
cmp   byte ptr [bx+1]     ,'u'
jnz   cmd2                ; Не u
; Освобождение блока памяти
; Проверка загруженности
mov   ax,0ff00h
int   2fh
cmp   ax,0ffh
jz    uninst ; Программа в памяти
lea   dx,msgno    ; Вывод сообщения о том,
mov   ah,9        ; что программы нет в памяти
int   21h
int   20h
uninst: call  set_int       ; Восстановление векторов прерываний
int   20h
cmd1: inc     bx
loop  cmd
cmd2: mov     ax,0ff00h     ; Проверка загруженности
      int     2fh
cmp   ax,0ffh
jnz   first_start     ; Не установлена
lea   dx,msg          ; Вывод сообщения о том,
mov   ah,9            ; что драйвер уже загружен
int   21h
int   20h
first_start:    mov   ax,3505h ; Сохранение старого вектора прерывания 5
      int     21h
      mov     cs:old_int5_off,bx
      mov     cs:old_int5_seg,es
```

```
        mov   ax,2505h              ; Функция 25h, вектор 5
        lea   dx,new_int5
        int   21h                   ; Запись нового вектора 5
        mov   ax,352fh ; Сохранение старого вектора прерывания 2Fh
        int   21h
        mov   cs:old_int2F_off,bx
        mov   cs:old_int2F_seg,es
        lea   dx,new_int2F          ; Запись нового вектора прерывания 2Fh
        mov   ax,252fh
        int   21h
        mov   ax,351ch ; Сохранение старого вектора прерывания 8
        int   21h
        mov   cs:old_int8_off,bx
        mov   cs:old_int8_seg,es
        lea   dx,new_int8           ; Запись нового вектора прерывания 8
        mov   ax,251ch
        int   21h
        mov   ah,34h ; Запись указателя на флаг критической секции DOS
        int   21h
        mov   word ptr _crit,bx
        mov   word ptr _crit[2],es
; Определение адреса видеобуфера
        mov   ah,0fh          ; Функция получения видеорежима
        int   10h
        cmp   al,7
        jz    ini1
        mov   vbuf,0b800h
ini1:           lea     dx,msg2
        mov   ah,9
        int   21h
; Завершение программы с оставлением резидентной части в памяти
        mov   dx,(ressize+10fh)/16
        mov   ax,3100h
        int   21h
init            endp
set_int         proc    near
        mov   ax,3505h
        int   21h     ; ES – сегментный адрес PSP резидента
        mov   adr_psp,es
; Восстановление старого вектора 2Fh
        push  ds
        mov   dx,es:old_int2F_off
        mov   ax,es:old_int2F_seg
        mov   ds,ax
        mov   ax,252fh       ; Установка старого вектора 2Fh
        int   21h
```

```
        mov   dx,es:old_int8_off
        mov   ax,es:old_int8_seg
        mov   ds,ax
        mov   ax,251ch      ; Установка старого вектора 8
        int   21h
        mov   dx,es:old_int5_off
        mov   ax,es:old_int5_seg
        mov   ds,ax
        mov   ax,2505h      ; Установка старого вектора 5
        int   21h
        pop   ds
        mov   ah,9
        lea   dx,msg1
        int   21h
        mov   es,adr_psp
        mov   ah,49h        ; Освобождение памяти
        int   21h
        ret
set_int         endp
msg             db      0dh,0ah,'Программа уже в памяти',0dh,0ah,'$'
msgno           db      0dh,0ah,'Программы нет в памяти',0dh,0ah,'$'
msg1            db      0dh,0ah,'Программа выгружена',0dh,0ah,'$'
msg2            db      0dh,0ah
                db      'Программа для записи содержимого символьного',0dh,0ah
                db      'экрана в файл FILESCR&.TXT.',0dh,0ah
        db      'ALESOFT (C) Roshin A. 1994.',0dh,0ah
        db      'Для копирования нажамите PrtSc.',0dh,0ah
        db      'В файл можно записать не более 32 экранов.',0dh,0ah
        db      'Для выгрузки программы следует набрать'
        db      ' filescr /u',0dh,0ah
        db      0dh,0ah,'$'
Code  ENDS
END   resprog
```

1.7 Вопросы для самсопроверки

1. Распределение памяти в IBM-совместимых ЭВМ
2. Векторы прерываний
3. Функция взятия вектора прерывания
4. Функция установки вектора прерывания
5. Резидентная программа
6. Структура резидентной программы
7. Установка резидентной программы
8. Передача управления резидентной программе
9. Восстановление вектора прерывания

10.Перехват вектора прерывания
11.Защита резидентной программы от повторной загрузки
12.Процедура загрузки DOS
13.Выгрузка резидентной программы
14.Обработка командной строки
15.Размещение резидентной программы в памяти ЭВМ
16.Обработка ключа выгрузки
17.Определение размера резидентной части
18.Префикс программы
19.Установка резидентной программы
20.Определение адреса установленной резидентной программы
21.Специфика резидентных программ
22.Функция 31h int 21h
23.Прерывание 27h
24.Отличие резидентной программы от загружаемой
25.Специфика драйверов
26.Работа с блоками памяти
27.Процедура загрузки DOS
28.Код завершения
29.Файл config.sys
30.Цепочка драйверов DOS
31.Размещение резидентной программы в памяти ЭВМ

Тема 2 Драйверы устройств в среде MS-DOS

2.1 Введение в драйверы

Работа любой ЭВМ связана с более или менее (обычно более) частым обращением к внешним устройствам. При этом следует иметь в виду, что пользователь и сама ЭВМ обычно различным образом трактуют понятие "внешнее устройство". Пользователю чаще всего не приходит в голову, что жесткий диск, гибкий диск, дисплей, а тем более клавиатура – внешнее устройство с точки зрения ЭВМ. Да и само понятие ЭВМ может трактоваться различным образом. Для пользователя ЭВМ – это существо, которое взаимодействует с ним посредством дисплея и клавиатуры (иногда также с помощью микрофона, динамика, сканера и т.д.) и имеет внутри себя все, что необходимо для его функционирования (жесткий и гибкий диски, различные порты и пр.).

Системному программисту, однако, следует четко представлять себе, что ЭВМ – это аппаратная часть (процессор с необходимым окружением и

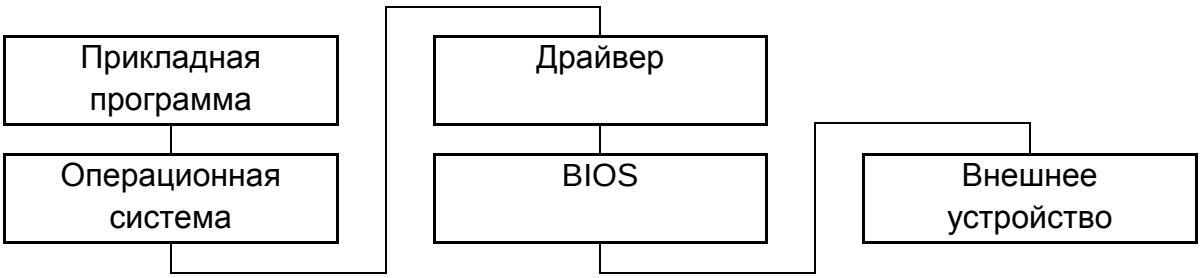
памятью), BIOS – базовая система ввода-вывода, жестко связанная с аппаратной частью (типом процессора, используемыми микросхемами и т. п., реализованная обычно в постоянном запоминающем устройстве – ПЗУ), и операционная система (MS-DOS, DR-DOS, OS-2, UNIX или что-то в этом духе). Пользователь (точнее – программа пользователя) обычно взаимодействует с операционной системой, т.е. с программой, предназначенной как раз для взаимодействия с пользователем.

Центральной частью операционной системы является ядро, занимающееся распределением памяти, управлением файловой системой и обработкой запросов к внешним устройствам.

Затем идет интерфейсная часть DOS, которая обеспечивает связь программ пользователя с операционной системой для взаимодействия с устройствами и дисковыми файлами, для обработки функций времени и даты, для управления видеорежимами и вывода на экран текста и графических образов, для ввода символов с клавиатуры и т.д.

Затем уже идут драйверы, которые взаимодействуют с внешними устройствами непосредственно или через BIOS.

Таким образом, взаимодействие программы пользователя с внешними устройствами обычно осуществляется по цепочке:



Для каждого подключенного к ЭВМ устройства имеется свой драйвер. Каждый запрос программы пользователя на обслуживание преобразуется DOS в последовательность простых команд драйвера и передает их соответствующему драйверу.

2.2 Драйвер устройства DOS

Устанавливаемый драйвер устройства – это программа в специальном формате, загружаемая в память во время загрузки DOS. Программа драйвера устройства состоит из следующих основных частей:

заголовка устройства,
рабочей области драйвера,
локальных процедур,
процедуры СТРАТЕГИЯ,
процедуры ПРЕРЫВАНИЕ,
программ обработки команд DOS.

Первой частью файла должна быть 18-байтовая структура – заголовок устройства, структура которого приведена ниже. Поле Next_Device, имеющее при загрузке значение смещения –1 (0ffffh) модифицируется DOS так, чтобы указывать на начало следующего драйвера в цепочке. DOS поддерживает связный список драйверов, начиная с устройства НУЛЬ (nul:). Драйвер устройства НУЛЬ находится в списке первым и содержит указатель на следующий драйвер. Каждый следующий драйвер содержит такой же указатель, значение которого в последнем драйвере равно -1. Каждый драйвер содержит имя своего устройства, по которому DOS и находит нужный драйвер.

Заголовок драйвера устройства

Значение поля Next_Device для последнего устройства в цепочке принимает значение –1 (0ffffh).

Поле "Имя устройства" содержит 8-символьное имя для символьного устройства или количество обслуживаемых устройств – для блоковых.

Поле DevAttr Заголовка устройства указывает свойства устройства. Ниже приведены значения отдельных разрядов слова состояния.

Бит		Маска
0	1 = стандартное входное устройство	0001h
1	1 = стандартное выходное устройство	0002h
2	1 = стандартное устройство NUL	0004h
3	1 = часы	0008h
6	1 = поддерживает логические устройства	0040h
11	1 = поддерживает open/close/RM	0800h
13	1 = не IBM блочное устройство	2000h
14	1 = поддерживает IOCTL	4000h
15	1 = символьное устройство; 0 = блочное устройство	8000h

Замечания:

устройство NUL не может быть переназначено

бит устройства не-IBM влияет на обработку запроса "построить блок BPB"

бит символьного устройства влияет на запросы ввода и вывода и определяет смысл поля 'имя устройства' в Заголовке устройства. Если этот бит равен 0, устройство является блочным устройством (обычно дисковод)

бит часов указывает на замещение устройства CLOCK$. CLOCK$ – это символьное устройство, обрабатывающее запросы устройства на ввод и вывод длиной ровно в 6 байтов. Запрос на ввод (код команды 4) должен возвратить 6 байтов, указывающих текущие время и дату. Запрос на вывод (код команды 8) должен принимать 6 байтов, содержащих значения часов и календаря.

При обращении к драйверу DOS формирует запрос устройства, в котором указывается, какую команду должен выполнить драйвер, а также передаются параметры команды, если это необходимо. Команды, используемые при вызове устройств в MS-DOS, приведены ниже.

Команда	Наименование
0	Инициализировать устройство
1	Контроль носителя
2	Построить BPB
3	IOCTL ввод
4	Ввод (читать с устройства)
5	Неразрушающий ввод
6	Статус ввода
7	Сброс ввода
8	Вывод (писать на устройство)
9	Вывод с верификацией
0ah	Статус вывода
0bh	Сброс вывода
0ch	IOCTL вывод
0dh	Открыть устройство
0eh	Закрыть устройство
0fh	Съемный носитель
13h	Общий запрос IOCTL
17h	Дать логическое устройство
18h	Установить логическое устройство

2.3 Описание команд драйвера

0 Инициализация

Первая команда, выдаваемая в драйвер диска после загрузки. Она служит для настройки драйвера и для получения следующих сведений:

сколько накопителей поддерживает драйвер

адрес конца драйвера

адрес таблицы BPB (количество BPB по числу поддерживаемых накопителей).

1 Контроль носителя

Эта команда всегда вызывается до дисковых операций считывания и записи для проверки смены носителя. Варианты ответа драйвера на запрос:

носитель не сменялся

носитель был сменен

не знаю

2 Получение BPB

Эта команда выдается в драйвер, если была определена смена носителя. Для жестких дисков команда получения BPB вызывается только один раз.

Если драйвер в ответ на контроль смены носителя отвечает "не знаю", вызывается команда получения BPB, если в DOS нет "грязных" буферов, т.е. буферов, в которых содержатся модифицированные данные, еще не записанные на диск. Если "грязные буферы" есть, DOS считает, что носитель сменен не был.

По команде получения BPB драйвер должен прочесть с диска загрузочный сектор, где по смещению 11 находится BPB. BPB помещается в рабочую область DOS, и драйвер возвращает DOS указатель на BPB.

3 IOCTL-ввод

Для блоковых устройств эта команда несущественна

4 Ввод

DOS передает драйверу количество считываемых секторов, номер начального сектора и область передачи данных, в которую помещаются считываемые данные. DOS должна заранее прочесть FAT и каталог для определения номеров требуемых секторов. Номер начального сектора отсчитывается от нуля от начала дискеты или раздела жесткого диска. Драйвер диска должен преобразовать логический номер начального сектора в физические параметры – дорожку, головку и физический номер сектора на дорожке.

8 Вывод

Это команда для записи одного или нескольких секторов на диск. Она аналогична команде ввода, но инверсна по направлению передачи данных.

9 Вывод с контролем

Эта команда аналогична команде вывода, но после записи данных драйвер осуществляет их считывание и проверку.

Команда VERIFY DOS устанавливает флажок VERIFY с состояние включено (ON) или выключено (OFF). В состоянии "включено" все команда

записи на диск передаются драйверу как команды вывода с контролем. Драйвер может сам осуществлять контроль состояния флажка VERIFY, дублируя его в своей переменной.

10 Статус ввода

Проверяет состояние устройства. Если устройство не готово, на него нельзя подавать команды ввода или вывода.

11 Сброс ввода

Очищает любой ввод, накопленный в буфере устройства. Используется, например, при ожидании подтверждения критических операций.

12 IOCTL-вывод

Эта команда может использоваться для посылки управляющей информации в драйвер. Однако для реализации специальных функций драйвера надо разрабатывать специальную программу.

13 Открытие устройства

Эта команда сообщает драйверу о появлении файла, открытого для записи или чтения. Драйвер может учитывать наличие открытых файлов перед выполнением операций чтения и записи. Для обработки команды открытия и закрытия устройства в слове атрибутов заголовка устройства должен быть установлен бит открытия/закрытия/сменный.

14 Закрытие устройства

Эта команда выдается в драйвер, когда программа закрывает устройство (при закрытии файла на диске).

По командам открытия и закрытия устройства драйвер может подсчитывать количество открытых файлов. При отсутствии открытых файлов драйвер может блокировать операции ввода и вывода.

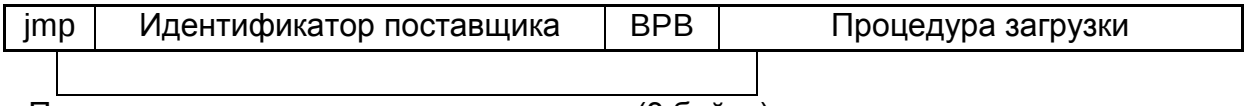
15 Сменный носитель

Эта команда позволяет запросить драйвер устройства, является ли носитель сменным. (В случае несменного носителя программа может считать, что смены диска не было.) Эта команда выдается в драйвер, если в слове атрибутов заголовка устройства установлен бит открытия/закрытия/сменный.

2.4 Создание драйверов блочных устройств

Для написания драйвера блочного устройства (обычно это диски) необходимо хорошо представлять себе структуру данных этого блочного устройства. Ниже рассматривается листинг драйвера блочного устройства – драйвер RAM-диска. Для корректного написания такого драйвера рассмотрим сначала структуру данных диска.

Загрузочная запись имеется на любом диске и размещается в начальном секторе. Она состоит из команды перехода на программу начальной загрузки, идентификатора поставщика, блока параметров BIOS (BPB) и программы начальной загрузки.



Переход на начало программы загрузки (3 байта)

- Идентификатор поставщика (8 байтов) – DOS не используется. Обычно здесь обозначена фирма и версия DOS.
- BPB (BIOS Parameter Block – 19 байтов) – информация о диске для DOS.
- Процедура загрузки соджержит коды программы начальной загрузки, которая загружается в память и получает управление. Она загружает резидентные драйверы, формирует связный список драйверов, анализирует содержимое файла CONFIG.SYS, загружает (если находит)
- описанные в нем драйверы, находит и загружает резидентную часть COMMAND.COM и передает управление ей. Дальнейшая загрузка осуществляется уже программой COMMAND.COM.

Блок параметров BIOS

Смещение	*Размер*	*Обозначение*	*Содержание поля*
+0	2	sect_siz	Размер сектора в байтах
+2	1	clus_siz	Число секторов в кластере
+3	3	res_sect	Количество зарезервированных секторов
+5	1	fat_num	Количество FAT на диске
+6	2	root_siz	Размер каталога (количествово файлов в корневом каталоге)
+8	2	num_sect	Общее количество секторов
+10	1	med_desc	Дескриптор носителя
+11	2	fat_size	Число секторов в FAT
+13	2	sec_trac	Число секторов на дорожке
+15	2	num_had	Число головок
+17	2	hidd_sec	Число скрытых секторов

- Размер сектора sect_siz – содержит число байтов в секторе для данного носителя. Допустимые размеры секторооов:128, 256, 512 и 1024 байтов.
- Число секторов в кластере – clus_siz определяет количество секторов в минимальной единице распределения дискового пространства.

– Количество зарезервированных секторов – res_sect показывает, сколько секторов зарезервировано для загрузочной записи. Обычно это поле содержит 1.
– Количество FAT на диске – fat_num указывает количество копий FAT на диске (обычно 2).
– Размер каталога root_siz – указывает максимальное количество файлов в корневом каталоге. Каждый элемент каталога занимает 32 байта, сектор содержит 16 элементов каталога.
– Общее количество секторов num_sect – общий размер диска в секторах. Это число должно включать секторы загрузочной записи, двух FAT, каталога и области данных пользователя. Для жестких дисков это число равно значению в последнем элементе таблицы разделов.
– Дескриптор носителя – описывает диск для MS-DOS:

F8h – жесткий диск
F9h – двухсторонний ГМД 5,25" (15 секторов)
двухсторонний ГМД 3,5"
FAh – RAM – диск
FCh – односторонний ГМД 5,25" (9 секторов)
двухсторонний ГМД 8" (одинарная плотность)
FDh – двухсторонний ГМД 5,25" (9 секторов)
FEh – односторонний ГМД 5,25" (8 секторов)
односторонний ГМД 8" (одинарная плотность)
односторонний ГМД 8" (двойная плотность)
FFh – двухсторонний ГМД 5,25" (8 секторов)

– Число секторов в FAT – fat_size число секторов в каждой FAT.
– Число секторов на дорожке sec_trac – стандартные значения для ГМД – 8, 9 и 15, – для ЖМД – 17.
– Число головок num_had – 1 или 2 для ГМД, для ЖМД – много.
– Число скрытых секторов hidd_sec – количество секторов, предшествующих активному разделу. Это смещение, которое добавляется к смещению файла внутри активного раздела для получения физического расположения файла на диске.

Сектор разделов	Раздел 1	Раздел 2	Раздел 3	Раздел 4
Скрытые секторы для раздела 4				
Скрытые секторы для раздела 3				
Скрытые секторы для раздела 2				

Скрытые секторы для раздела 2

Таблица размещения файлов FAT содержит информацию об использовании дискового пространства файлами. В FAT имеется элемент для каждого доступного кластера.

12-битный элемент	16-битный элемент	Значение
000h	0000h	Свободен
001h-FEFh	0001h-FFEFh	Занят
FF0h-FF6h	FFF0h-FFF6h	Зарезервирован
FF7h	FFF7h	Дефективен
FF8h-FFFh	FFF8h-FFFFh	Конец цепи кластеров

Доступное пользователю пространство начинается с первого свободного кластера, имеющего номер 2. Число файлов в каталоге зависит от типа диска:

Элементов каталога	Секторов каталога	Тип диска
64	4	Односторонние ГМД
112	7	Двухсторонние ГМД
224	14	ГМД большой емкости
512	32	Жесткие диски

Смещение	Размер	Содержание
+0	8	Имя файла
+8	3	Расширение имени файла
+11	1	Атрибуты файла
+12	10	Резерв DOS
+22	2	Время создания или последней модификации
+24	2	Дата создания или последней модификации
+26	2	Начальный кластер
+28	4	Размер файла

- Имя файла – до 8 символов, выравнивается по левому краю, незанятые позиции заполняются пробелами. При удалении файла первый байт имени заменяется кодом E5h. Пока элемент каталога не использован, первый байт имени файла содержит 00h. DOS прекращает просмотр каталога, как только встретит значение 00h в первом байте имени файла. При обнаружении на этом месте кода E5h просмотр продолжается.
- Расширение имени файла – до 3 символов, выровненных по левому краю. Необязательно.

– Начальный кластер – номер первого кластера, распределенного файлу

– Размер файла – в байтах (4-байтовое значение)

– Атрибуты файла

Код	Значение
00h	Обыкновенный файл
01h	Файл только для чтения
02h	Скрытый файл
04h	Системный файл
08h	Метка тома
10h	Подкаталог
20h	Архивный бит

– Время создания или последней модификации подкаталога не изменяется при добавлении в подкаталог нового элемента. (То же относится к дате.)

Поле времени	Поле	Смещение	Биты
	Часы	17h	7 – 3
	Минуты	17h	2 – 0
		16h	7 – 5
	Секунды	16h	4 – 0

Поле даты	Поле	Смещение	Биты
	Год	19h	7 – 1
	Месяц	19h	0
		18h	7 – 5
	Число	18h	4 – 0

Год – относительно 1980 года.

2.5 Драйвер RAM-диска

Состоит из собственно драйвера и пространства памяти для диска. Из четырех частей загрузочной записи для RAM-диска будут реализованы только две – идентификацию поставщика и BPB. В BPB задаются размер RAM-диска (100К), размер FAT и размер каталога. BPB RAM-диска:

Смещение	Размер	Имя	Значение	Содержание
+0	2	sect_siz	512	Размер сектора в байтах
+2	1	clus_siz	1	Число секторов в кластере
+3	2	res_sect	1	Количество зарезервированных секторов

+5	1	fat_num	1	Количество FAT на диске
+6	2	root_siz	48	Размер корневого каталога (число файлов)
+8	2	num_sect	205	Общее количество секторов
+10	1	med_desc	FEh	Дескриптор носителя
+11	2	fat_size	1	Число секторов в FAT
+13	2	sec_trac	0	Число секторов на дорожке
+15	2	num_had	0	Число головок
+17	2	hidd_sec	0	Число скрытых секторов

Всего секторов	
1	для загрузочной записи
1	для 1 FAT (1.5 байта * 200 кластеров = 300 байтов)
3	для каталога (32 байта * 48 файлов = 1536 байтов)
200	для данных (100 KDB)
205	секторов на RAM диске

Ниже приведен текст драйвера RAM-диска.

```
; Заголовок
; Драйвер RAM-диска со звуковым сигналом

; Инструкции ассемблеру
code  segment  para  puublic
ramdisk      proc   far
assume cs:code, ds:code, es:code
; Структура заголовков запросов
rh      struc ; Фиксироованная структура заголовка
rh_len       db     ?      ; Длина пакета
rh_unit      db     ?      ; Номер устройства
rh_cmd       db     ?      ; Команда
rh_status    dw     ?      ; Возвращается драйвером
rh_res1      dd     ?      ; Резерв
rh_res2      dd     ?      ; Резерв
rh      ends
; Инициализация
rh0     struc  ; Заголовок запроса команды 0
rh0_rh       db     size rh dup(?)       ; Фиксированная часть
rh0_nunits   db     ?                    ; Число устройств в группе
rho_brk_ofs  dw     ?                    ; Смещение конца драйвера
rho_brk_seg  dw     ?                    ; Сегмент конца драйвера
rh0_bpb_tbo  dw     ?                    ; Смещение указателя массива BPB
rh0_bpb_tbs  dw     ?                    ; Сегмент указателя массива BPB
rh0_drv_ltr  db     ?                    ; Первый доступный накопитель
rh0     ends
; Проверка смены носителя
```

```
rh1     struc                   ; 33 для команды 1
rh1_rh          db      size rh dup(?)
rh1_media       db      ?       ; Дескриптор носителя из DPB
rh1_md_stat db      ?       ; Возвращаемое драйвером
rh1     ends                    ; состояние носителя
; Построить блок BPB
rh2     struc                   ; 33 для команды 2
rh2_rh          db      size rh dup(?)
rh2_media       db      ?       ; Дескриптор носителя из DPB
rh2_buf_ofs     dw      ?       ; Смещение DTA
rh2_buf_seg dw      ?       ; Сегмент DTA
rh2_pbpbo       dw      ?       ; Смещение указателя BPB
rh2_pbpbs       dw      ?       ; Сегмент указателя BPB
rh2     ends
; Запись
rh4     struc
rh4_rh          db      size rh dup(?)
rh4_media       db      ?       ; Дескриптор носителя из DPB
rh4_buf_ofs     dw      ?       ; Смещение DTA
rh4_buf_seg dw      ?       ; Сегмент DTA
rh4_cont        dw      ?       ; Счетчик передачи
rh4_start       dw      ?       ; Начальный сектор
; Запись
rh8     struc
rh8_rh4         db      size rh4 dup(?)     ; Совпадает с командой
rh8     ends                                ; чтения
; Запись с контролем
rh9     struc                               ; Совпадает с
rh9_rh4         db      size rh4 dup(?)     ; командой чтения
; Проверка сменяемости диска
rh15    struc                               ; Состоит
rh15_rh         db      size rh dup(?)      ; только из заголовка
; Основная процедура
begin:
start_address       equ     $       ; Начальный адрес драйвера
; Этот адрес нужен для последующего определения начала области данных
; Заголовок устройства для DOS
next_dev        dd      -1              ; Других драйверов нет
attribute       dw      2000h           ; Блоковое, формат не IBM
strategy        dw      dev_strategy    ; Адрес процедуры СТРАТЕГИЯ
interrpt        dw      dev_interrpt    ; Адрес процедуры ПРЕРЫВАНИЕ
dev_name        db      1               ; Число блоковых устройств
                db      7 dup(?)        ; Дополнение до 7 бит
; Атрибуты – сброшен бит 15 – блоковые, установлен бит 13 – не формат IBM
; (DOS не будет использовать байт дескриптора носителя для определения
```

```
; размера диска)
; Имя – DOS не пзволяет драйверам блоковых устройств иметь имена.
; Значение первого байта этого поля равно числу RAM-дисков, которыми будет ;
управлять этот драйвер. 1 здесь сообщает DOS, что имеется только один
; RAM-диск.
; Рабочее пространство драйвера
rh_ofs      dw    ?       ; Смещение заголовка запроса
rh_seg      dw    ?       ; Сегмент заголовка запроса
; Переменные для адреса заголовка запроса, который DOS
; передает драйверу при вызове процедуры СТРАТЕГИЯ
boot_rec    equ   $             ; Начало загрузочной записи
        db    3 dup(0)    ; Вместо команды перехода
        db    'MIP 1.0 '  ; Идентификатор поставщика
bpb         equ   $             ; Начало BPB
bpb_ss      dw    512           ; Размер сектора 512 байтов
bpb_cs      db    1             ; 1 сектор в кластере
bpb_rs      dw    1             ; 1 зарезервированный сектор
bpb_fn      db    1             ; 1 FAT
bpb_ros     dw    48            ; 48 файлов в каталоге
bpb_ns      dw    205           ; Общее кол-во секторов
bpb_md      db    0feh          ; Дескриптор носителя
bpb_fs      dw    1             ; Число секторов в FAT
bpb_ptr     dw    bpb   ; Указатель BPB
; Текущая информация о параметрах операции с диском
total       dw    ?       ; Счетчик секторов для передачи
verify      db    0       ; Контроль: 1 – вкл. 0 – нет
start       dw    0       ; Номер начального сектора
disk        dw    0       ; Начальный параграф RAM-диска
buf_ofs     dw    ?       ; Смещение DTA
buf_seg     dw    ?       ; Сегмент DTA
res_cnt     dw    5       ; Число зарезервированных секторов
ram_par     dw    6560  ; Параграфов памяти
bell        db    1       ; 1 – звуковой сигнал при обращении
; Зарезервированные секторы – загрузочная запись, FAT и каталог
; Процедура СТРАТЕГИЯ
dev_strategy:       mov   cs:rh_seg,es
                    mov   cs:rh_ofs,bx
                ret
; Процедура ПРЕРЫВАНИЕ
dev_interrupt:      cld
                push  ds
                push  es
                push  ax
                push  bx
                push  cx
```

```
                push   dx
                push   di
                push   si
        mov    ax,cs:rh_seg ; Восстановление ES и BX,
        mov    es,ax        ; сохраненных при вызове
        mov    bx,cs:rh_ofs ; процедуры СТРАТЕГИЯ
; Переход к подпрограмме обработки соответствующей команды
mov     al,es:[bx].rh_cmd   ;Команда из загол.запроса
rol     al,1                ; Удвоение
lea     di,cmdtab           ; Адрес таблицы переходов
xor     ah,ah
add     di,ax
jmp     word ptr[di]
; Таблица переходов для обработки команд
cmdtab          dw      INITIALIZATION      ; Инициализация
        dw      MEDIA_CHECK     ; Контооль носителя (блоков.)
                dw      GET_BPB             ; Получение BPB
        dw      IOCTL_INPUT     ; IOCTL-ввод
        dw      INPUT           ; Ввод
        dw      ND_INPUT        ; Неразрушающий ввод
        dw      INPUT_STATUS    ; Состояние ввода
        dw      INPUT_CLEAR     ; Очистка ввода
        dw      OUTPUT          ; Вывод
        dw      OUTPUT_VERIFY   ; Вывод с контролем
        dw      OUTPUT_STATUS ; Состояние вывода
        dw      OUTPUT_CLEAR    ; Очистка вывода
        dw      IOCTL_OUT ; IOCTL-вывод
        dw      OPEN            ; Открытие устройства
        dw      CLOSE           ; Закрытие устройства
        dw      REMOVABLE       ; Сменный носитель
        dw      OUTPUT_BUSY     ; Вывод по занятости
; Локальные процедуры
save    proc    near    ; Сохраняет данные из заголовка запроса
; Вызывается командами чтения и записи
mov     ax,es:[bx].rh4_buf_seg      ; Сохранение
mov     cs:buf_seg,ax               ; сегмента DTA
mov     ax,es:[bx].rh4_buf_ofs      ; Сохранение
mov     cs:buf_ofs,ax               ; смещения DTA
mov     ax,es:[bx].rh4_start        ;Сохранение номера
mov     cs:start,ax                 ;начального сектора
mov     ax,es:[bx].rh4_count
xor     ah,ah                       ; На всякий случай
mov     cs:total,ax                 ; Кол-во перед. секторов
ret
save    endp
```

```
; Процедура вычисления адреса памяти
; Вход: cs:start – начальный сектор
; cs:total – кол-во передаваемых секторов
; cs:disk – начальный адрес RAM-диска
; Возврат: ds – сегмент
; cs – число передаваемых данных
; SI=0
; Использует AX, CX, SI, DS
calc    proc    near
mov     ax,cs:start     ; Номер начального сектора
mov     cl,5            ; Умножить на 32
shl     ax,cl           ; Номер начального параграфа
mov     cx,cs:disk      ; Нач. сегмент RAM-диска
add     cx,ax           ; Абс. нач. парараф (сегмент)
mov     ds,cx           ; DS = начальный сегмент
xor     si,si           ; SI = 0
mov     ax,cs:total     ; Количество передаваемых секторов
cmp     ax,129          ; Должно быть не более 128
jc      calc1           ; < 129 ( < 64 KB )
mov     ax,128          ; Принудительно = 128 сект.
calc1: mov    cx,512          ; Байтов в секторе
mul     cx              ; AX = число перед. байтов
mov     cx,ax           ; Пересылка в CX
ret
calc    endp
; Включение звука (если надо)
bell1   proc    near
test    byte ptr bell,0ffh      ; Звук нужен ?
jz      nobell          ; Не нужен
mov     al,0b6h                 ; Управляющее слово
out     43h,al                  ; Посылка его в РУС
mov     ax,400h                 ; Коэффициент деления
out     42h,al                  ; Мл. байт в канал 2
xchg    al,ah
out     42h,al                  ; Ст. байт в канал 2
in      al,61h                  ; Чтение порта динамика
or      al,3                    ; Включение динамика
out     61h,al
nobell:         ret
bell1   endp
; Выключение звука (без проверки необходимости)
bell2   proc    near
in      al,61h          ; Порт динамика
and     al,0fch         ; Выключение динамика
out     61h,al
ret
```

```
bell2   endp
; Обработка команд DOS
; Команда 0: Инициализация
initialization: call    bell1   ; Включение звука
        call    initial ; Вывод сообщения
        push    cs
        pop     dx      ; DX = CS
; Определение конца RAM-диска
        lea     ax,cs:start_disk        ;Отн.нач.адр.RAM-диска
        mov     cl,4                    ; Деление на 16
        ror     ax,cl                   ; Отн.нач.параграф RAM-диска
        add     dx,ax                   ; Абсолютн. нач. парагр. диска
        mov     cs:disk,dx              ;Сохранение абс. нач. параграфа
        add     dx,ram_par              ; + размер диска в параграфах
; Возврат в DOS адреса конца
        mov     es:[bx].rh0_brk_ofs,0       ; Смещение = 0
        mov     es:[bx].rh0_brk_seg,dx      ; Семент
; Возврат числа устройств в блоковом устройстве
        mov     es:[bx].rh0_nunits,1        ; 1 диск
; Возврат адреса BPB (одного)
        lea     dx,bpb_ptr                  ; Адрес указателя
        mov     es:[bx].rh0_bpb_tbo,dx      ; Смещение
        mov     es:[bx],rh0_bpb_tbs,cs      ; Сегмент
; Инициализация загрузочной записи, FAT и каталога
        push    ds                          ; CALC портит DS
        mov     cs:start,0                  ; Нач. сектор = 0
        mov     ax,cs:res_cnt               ; Кол-во зарезерв. сект.
        mov     cs:total,ax                 ; Кол-во передав. сект.
        call    calc            ; Вычисл. физич. параметров
        mov     al,0            ; Чем заполнять
        push    ds              ; DS – начало RAM-диска
        pop     es
        mov     di,si           ; DI = SI = 0
        rep     stosb           ; Заполн. зарезерв. сект. нулями
        pop     ds              ; Восстановление DS = CS
; Загрузочная запись
        mov     es,cs:disk      ; Начальный сегмент диска
        xor     di,di
        lea     si,cs:boot_rec      ;Смещение загруз. записи
        mov     cx,30               ;Кол-во копируемых байтов
        rep     movsb               ;Копирование
; Создать 1 FAT
        mov     cs:start,1          ; Логич. сектор 1
        mov     cs:total,1          ; Не имеет значения
        call    calc                ; Установка DS:SI
```

```
        mov   ds:word ptr [si],0feffh      ;Зап. в FAT дес криптора
        mov   ds:word ptr 1[si],0ffffh     ; носи теля FEh и 5 байтов FFh
        mov   ds:word ptr 3[si],0ffffh
        call  bell2                        ; Выключение звука
; Восстановление ES:BX
        mov   ax,cs:rh_seg
        mov   es,ax
        mov   bx,cs:rh_ofs
        jmp   done                 ; Выход с уст. бита "сделано"
; Команда 1: Контроль носителя
; -1 – носитель сменен, 0 – не знаю, +1 – носитель не менялся
; Для жестких и RAM-дисков всегда +1
media_check:    mov   es:[bx].rh1_media,1
                jmp   done
; Команда 2: Получение BPB
; Обработчик команды считывает BPB из RAM-диска в буфер
;данных, определенный DOS. Адрес массива BPB передается DOS
;в заголовке запроса get_bpb:
; Считывание загрузочной записи
push  es     ; Сохранение смещ. и сегм. заголовка запроса
push  bx
        mov   cs:start,0     ; Сектор 0
        mov   cs:total,1     ; Один сектор
        call  calc
push  cs
pop   es             ; ES = CS
lea   di,cs:bpb      ; Адрес BPB
add   si,11          ; 11 – смещение BPB
mov   cx,13          ; Длина BPB
rep   movsb
pop   bx
pop   es
mov   dx,cs:bpb_ptr              ; Указатель массива BPB
mov   es:[bx].rh2_bpbbo,dx       ; в заголовок запр.
mov   es:[bx].rh2_bpbbs,cs       ; Сегмент тоже
lea   dx,cs:bpb                  ; Адрес BPB
mov   es:[bx].rh2_buf_ofs,dx     ;Смещ. буф.= адр. BPB
mov   es:[bx].rh2_buf_seg,cs     ;Сегмент тоже
jmp   done           ; Выйти с взведенным битом "сделано"
; Команда 3: IOCTL-ввод
ioctl_input:  jmp   unknown      ; Выйти с уст. битом "ошибка"
; Команда 4: Ввод
; Эта команда передает драйверу номер начального сектора и
; количество считываемых секторов.
; Драйвер преобразует эти данные в физические адрес и
; размер и считывает данные из RAM-диска в буфер DOS.
```

```
input:  call    bell1   ; Включение звука (если разрешено)
call    save    ; Сохранене данных заголовка запроса
call    calc    ; Определение физического рач. адреса
mov     es,cs:buf_seg       ; ES:DI – адрес буфера
mov     di,cs:buf_ofs
mov     ax,di
add     ax,cx           ; Смещение + длина передачи
jnc     input1          ; Переход, если нет переполн.
mov     ax,0ffffh       ; Коррекция CX так, чтобы не
sub     ax,di           ; возникал выход за
mov     cx,ax           ; пределы сегмента
input1: rep     movsb       ; Считывание данных в буфер
call    bell2           ; Выключение звука
mov     es,cs:rh_seg ; Восстановление
mov     bx,cs:rh_ofs  ; ES и BX
jmp     done            ; Выйти с уст. битом "сделано"
; Команды 5, 6 и 7 не обрабаьываются драйверами блоковых
; устройств
; Команда 5: Неразрушающий ввод
nd_input:       jmp     busy   ; Выйти с уст. битом "занят"
; Команда 6: Состояние ввода
input_status: jmp     done   ; Выйти с уст. битом "сделано"
; Команда 7: Очистка ввода
input_clear:    jmp     done   ; Выйти с уст. битом "сделано"
; Команда 8: Вывод
; Драйвер пересчитывает номер сектора и количество переда-
; ваемых секторов в физический адрес начала и количество пе-
; редаваемых байтов, после чего заданное количество байтов
; записывается из буфера DOS в RAM-диск
output:         call    bell1   ; Включение звука (если разрешено)
        call    save    ; Сохранене данных заголовка запроса
        call    calc    ; Определение физического адреса
        push    ds
        pop     es      ; ES = DS
        mov     di,si   ; ES:DI = DS:SI (адр. RAM-диска)
        mov     ds,cs:buf_seg       ; DS:SI – адрес буфера с
        mov     si,cs:buf_ofs       ; записываемыми данными
        mov     ax,si
        add     ax,cx           ; Смещение + длина передачи
        jnc     output1         ; Переход, если нет переполн.
        mov     ax,0ffffh       ; Коррекция CX так, чтобы не
        sub     ax,si           ; возникал выход за
        mov     cx,ax           ; пределы сегмента
input1:         rep     movsb       ; Считывание данных в буфер
        mov     es,cs:rh_seg ; Восстановление ES:BX из-за
        mov     bx,cs:rh_ofs  ; возможного перехода к вводу
```

```
        cmp   cs:verify,0    ; Нужна проверка ?
        jz    output2        ; Нет
        mov   cs:verify,0    ; Сброс флага проверки
        jmp   input          ; Считать те же секторы
output2:    call  bell2          ; Выключить звук
        mov   es,cs:rh_seg ; Восстановление
        mov   bx,cs:rh_ofs  ; ES:BX
        jmp   done           ; Выйти с уст. битом "сделано"
; Команда 9: Вывод с контролем
; Устанавливает флаг контроля VERIFY и передает управление
; команде "вывод"
output_verify: mov  cs:verify,1    ; Установка флага контроля
        jmp   outpt          ; Переход на "вывод"
; Команды 10 (состояние вывода) и 11 (очистка вывода) пред-
; назначены только для символьных устройств.
; Команды 12 (IOCTL-вывод), 13 (открытия устройства) и 14
; (закрытия устройства) не обрабатываются в данном драйвере
; Команда 10: Состояние вывода
output_status:      jmp   done  ; Выйти с уст. битом "сделано"
; Команда 11: Очистка вывода
output_ckear:       jmp   done  ; Выйти с уст. битом "сделано"
; Команда 12: IOCTL-вывод
ioctl_output:       jmp   unknown     ; Выйти с уст. битом "ошибка"
; Команда 13: Открытие
open:       jmp   done  ; Выйти с уст. битом "сделано"
; Команда 14: Закрытие
close:      jmp   done  ; Выйти с уст. битом "сделано"
; Команда 15: Сменный носитель
; Драйвер по номеру устройства в группе, полученному от DOS,
; должен установить бит "занято" в слове состояния заголовка
; запроса в 1, если носитель не сменный, или в 0, если носи-
; тель сменный. в RAM-диске носитель несменный, поэтому сле-
; дует установить этот бит в 1.
removable:  mov   es:[bx].rh_status,200h     ; Установка бита "занято"
        jmp   done           ; Выйти с уст. битом "сделано"
; Команда 16: Вывод по занятости
; Это команда для символьных устройств. Данный драйвер должен
; установить бит 2ошибка" и код ошибки 3 (неизвестная команда)
output_busy:        jmp   unknown     ; Выйти с уст. битом "ошибка"
; Выход по ошибке
unknow:     or    es:[bx].rh_status,8003h    ; Уст. бита и кода ош.
        jmp   done                       ; Выйти с уст. битом "сделано"
; Обычный выход
done:       or    es:[bx].rh_status,100h     ; Уст. бит "сделано"
        pop   si
```

```
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        pop     es
        pop     ds
        ret                     ; Возврат в DOS
; Конец программы
end_of_program:
; Выравнивание начало RAM-диска на границу параграфа
if ($-start_address)mod 16 (если не 0)
  org ($-start_address)+(16-($-atart_address)mod 16)
endif
start_disk      equ     $
; Процедура initial помещается в начало RAM-диска, т.к.
; она выполняется единственный раз в команде нициализа-
; ции, после чего ее можно стереть.
initial  proc   near            ; Вывод сообщения на консоль
lea     dx,msg1         ; Адрес сообщения
mov     ah,9            ; Функция 9 – ввод строки
int     21h
ret
initial  endp
msg1  db      'RAMDISK driver',0dh,0ah,'$'
ramdisk         endp
code    ends
end     begin
```

2.6 Драйвер консоли

В качестве драйвера символьного устройства рассмотрим листинг драйвер консоли, предназначенный для замены стандартного драйвера. Такое предназначение драйвера предполагает, что драйвер должен выполнять, кроме специальных, еще и все функции стандартного драйвера. Ниже приведен листинг такого драйвера.

```
; Заголовок
; Драйвер консоли; назначение – заменить стандартный драйвер
; Инструкции ассемблеру
Code  segment  para  public
console         proc    far
assume cs:code, ds:code, es:code
; Стуктуры заголовка запроса
rh      struc   ; Структура заголовка
rh_len          db      ?       ; Длина пакета
```

```
rh_init         db      ?       ; Номер устройства (блоковые)
rh_cmd          db      ?       ; Команда драйвера устройства
rh_status       dw      ?       ; Возвращается драйвером
rh_res1         dd      ?       ; Резерв
rh_res2         dd      ?       ; Резерв
rh      ends
rh0     struc   ; Заголовок запроса команды 0
rh0_rh          db      size rh dup(?)          ; Фиксированная часть
rh0_numunit db      ?       ; Число устройств в группе
rh0_brk_ofs  dw     ?       ; Смещение конца
rh0_brk_seg dw      ?       ; Сегмент конца
rh0_bpb_pno  dw     ?       ; Смещение указ. массива BPB
rh0_bpb_pns  dw     ?       ; Сегмент указ. массива BPB
rh0_drv_itr    db     ?       ; Первый доступный накопитель
rh0     ends
rh4     struc   ; Заголовок запроса для команды 4
rh4_rh          db      size rh dup(?)          ; Фиксиоованная часть
rh4_media       db      ?       ; Дескриптор носителя из DPB
rh4_buf_ofs  dw     ?       ; Смещение DTA
rh4_buf_seg dw      ?       ; Сегмент DTA
rh4_count       dw      ?       ; Счетчик передачи (сект. -
rh4_start       dw      ?       ; Начальныйй сектор (блолоовые)
rh4     ends
rh5     struc   ; Заголовок запороса для команды 5
rh5_rh          db      size    rh dup(?)       ; Фиксированная часть
rh5_return      db      ?       ; Возвращаемый символ
rh5     ends
rh7     struc   ; Заголовок запроса для команды 7
rh7_len         db      ?       ; Длина пакета
rh7_unit        db      ?       ; Номер устройства (блоковые)
rh7_cmd         db      ?       ; Команда драйвера устройства
rh7_status      dw      ?       ; Возвращается драйвером
rh7_res1        dd      ?       ; Резерв
rh7_res2        dd      ?       ; Резерв
rh7     ends
rh8     struc   ; Заголовок запроса для команды 8
rh8_rh          db      size rh dup(?)          ; Фиксированная часть
rh8_media       db      ?       ; Дескриптор носителя из DPB
rh8_buf_ofs  dw     ?       ; Смещение DTA
rh8_buf_seg dw      ?       ; Сегмент DTA
rh8_count       dw      ?       ; Счетчик пер. (сект. –  блоковые, байтов – симв.)
rh8_start       dw      ?       ; Начальный сектор (блоковые)
rh8     ends
rh9     struc ; Заголовок запроса для команды 9
rh9_rh          db      size rh dup(?)          ; Фиксированная часть
rh9_media       db      ?                       ; Дескриптор носителя из DPB
```

```
rh9_buf_ofs  dw    ?       ; Смещение DTA
rh9_buf_seg  dw    ?       ; Сегмент DTA
rh9_count    dw    ?       ; Счетчик пер. (сект. –  блоковые, байты – символьные)
rh9_start    dw    ?       ; Начальный сектор (блоковые)
rh9    ends
```

```
; Основная процедура
```

```
start:
```

```
; Заголовок устройства для DOS
next_dev     dd    -1      ; Адес следующего устройства
attribute    dw    8003h ; Символьное, ввоод, вывод
strategy     dw    dev_strategy        ; Адр. проц. СТРАТЕГИЯ
interrupt    dw    dev_interrupt; Адр. проц. ПРЕРЫВАНИЕ
dev_name     db    'CON '              ; Имя драйвера
```

```
; Рабочее пространство для драйвера
rh_ofs       dw    ?       ; Смещение заголовка запроса
rh_seg       dw    ?       ; Сеггмент заголовка запроса
sav          db    0       ; Символ, считанный с клавиатуры
```

```
; Процедура СТРАТЕГИЯ (первый вызов из DOS)
; Это точка входа первого вызова драйвера. Эта процедура
;сохраняет адрес заголовка запроса в переменных rh_seg и rh_ofs.
```

```
; Процедура ПРЕРЫВАНИЕ (второй вызов из DOS)
; Осуществляет переход на обработку команды, номер которой
; находится в заголовке запроса. (То же, что и раньше.)
```

```
; Локальные процедуры (здесь одна)
tone   proc   near    ; В al – код символа
mov    ah,0
push   ax
mov    al,0b6h        ; Управляющее слово для таймера
out    43h,al         ; Посылка в РУС
mov    dx,0
mov    ax,14000       ; Частота
pop    cx             ; В CX – код символа
inc    cx             ; Вдруг в CX – нуль
div    cx             ; Деление 14000 на код символа
out    42h,al         ; Вывод в канал таймера мл. байта
xchg   ah,al          ; результата
out    42h,al         ; Выв. в канал тайм.ст.байта рез.
in     al,61h         ; Системный порт B
or     al,3           ; Включить динамик и таймер
out    61h,al
mov    cx,15000       ; Задержка
tone1: loop   tone1
in     al,61h
and    al,0fch        ; Выключение динамика и таймера
out    61h,al
```

```
ret
tone   endp
; Обработка команд DOS
; Команда 0 ИНИЦИАЛИЗАЦИЯ
initialization: call    initial  ; Вывод начального сообщения
        lea     ax,initial       ; Установка адреса конца
        mov     es:[bx].rh0_brk_ofs,ax      ; Смещение
        mov     es:[bx].rh0_brk_seg,cs      ; Сегмент
        jmp     done             ; Уст. бит СДЕЛАНО и выйти
; Команда 1 КОНТРОЛЬ НОСИТЕЛЯ
media_check:        jmp     done  ; Уст. бит СДЕЛАНО и выйти
; Команда 2 Получение BPB
get_bpb:            jmp     done  ; Уст. бит СДЕЛАНО и выйти
; Команда 3 Ввод IOCTL
ioctl_input:        jmp     unkn  ; Уст. бит ОШИБКА и выйти
; Команда 4 Ввод
input:  mov     cx,es:[bx].rh4_count        ;Загр. счетчик ввода
mov     di,es:[bx].rh4_buf_ofs      ; Смещение буфера
mov     ax,es:[bx].rh4_buf_seg      ; Сегмент буфера
mov     es,ax                       ; ES = сегмент буфера
read1:          xor     ax,ax
        xchg    al,sav          ; Взять сохраненный символ
        or      al,al           ; Он равен 0 ?
        jnz     read3           ; Нет – передать его в буфер
read2:                                  ; sav=0 – Вводить следующий символ
        xor     ah,ah           ; Функция 0 – считывание
        int     16h             ; Прерывание BIOS для клавиатуры
        or      ax,ax           ; 0 ? (буфер пуст)
        jz      read2           ; Взять следующий символ
        or      al,al           ; Это расширенная клавиша ?
        jnz     read3           ; Нет – передать ее код
        mov     sav,ah          ; Сохранить скан-код
read3:          mov     es:[di],al      ; Записать код в буфер
        inc     di              ; Сдвинуть указатель
        push    cx
        call    tone            ; (Портит CX)
        pop     cx
        loop    read1
        mov     es,cs:rh_seg ; Восстановить ES
        mov     bx,cs:rh_ofs  ; Восстановить BX
        jmp     done
; Команда 5 Неразрушающий ввод
nd_input:       mov     al,sav          ; Взять сохраненный символ
        or      al,al           ; = 0 ?
        jnz     nd1             ; Нет – возвратить его в DOS
```

```
        mov   ah,1          ; Функция BIOS контроль состояния
        int   16h
        jz    busy          ; (Z) – символов в буфере нет
nd1:    mov   es:[bx].rh5_return,al     ;Возвратить символ DOS
        jmp   done          ; Уст. бит СДЕЛАНО и выйти
; Команда 6 Состояние ввода
input_status: jmp  done     ; Установить бит СДЕЛАНО и выйти
; Команда 7 Очистка ввода
input_clear:  mov  sav,0    ; Сброс сохраненного символа
ic1:    mov   ah,1
        int   16h           ; BIOS – контроль сост. клавиатуры
        jz    done          ; (Z) – буфер пуст
        xor   ah,ah
        int   16h           ; BIOS Считывание символа
        jmp   ic1           ; Повторять до опустишения буфера
; Команда 8 Вывод
output: mov   cx,es:[bx].rh8_count      ;Взять счетчик вывода
        mov   di,es:[bx].rh8_buf_ofs    ;Смещение буфера
        mov   ax,es:[bx].rh8_buf_seg    ;Сегмент буфера
        mov   es,ax
        xor   bx,bx         ; (bl – цвет перед. плана в графике)
out1:   mov   al,es:[di]    ; Взять выводимый символ
        inc   di            ; Сместить указатель
        mov   ah,0eh        ; Вывод в режиме телетайпа
        int   10h
        loop  out1          ; Повторять (count) раз
        mov   es,cs:rh_seg ; Восстановление адреса
        mov   bx,cs:rh_ofs  ; заголовка запроса
        jmp   done
; Команда 9 Вывод с контролем
output_verify:      jmp   output
; Команда 10 Состояние вывода
output_status:      jmp   done
; Команда 11 Очистка вывода
output_clear:       jmp   done
; Команда 12 IOCTL-вывод
ioctl_out:          jmp   unkn  ; Установить бит ОШИБКА и выйти
; Команда 13 Открытие
open:               jmp   done
; Команда 14 Закрытие
close:              jmp   done
; Команда 15 Сменный носитель
removable:          jmp   unkn
; Команда 16 Вывод по занятости
output_busy:        jmp   unkn
```

```
; Выход по ошибке
unkn:  or      es:[bx].rh_status,8003h     ; Установить бит
       jmp     done                        ; ошибки и ее код
; Обычный выход
busy:  or      es:[bx].rh_status,200h      ;Установить бит ЗАНЯТ
done:  or      es:[bx].rh_status,100h      ;Уст. бит СДЕЛАНО
       pop     si
pop    si
pop    dx
pop    cx
pop    bx
pop    ax
pop    es
pop    ds
ret
; Конец программы
; Эта процедура вызывается только при инициализации
;и может быть затем стерта
initial proc   near
lea    dx,cs:msg1
mov    ah,9
int    21h    ; Вывод сообщения на экран
ret
initial endp
msg1   db              'Console driver',0dh,0ah,'$'
console        endp
Code   ends
End    start
```

2.7 Заключительные замечания

В заключение дадим некоторые рекомендации, к которым стоит прислушаться при написании и отладке драйверов.

- Для отладки и проверки драйверов всегда следует пользоваться тестовым загрузочным диском. Это:
  - изолирует проверку от стандартной рабочей среды
  - предотвращает "зависание" ЭВМ при загрузке DOS
- Драйвер должен начинаться с 0, а не с 100h
- Драйвер должен быть COM-программой.
  - в момент загрузки драйвера DOS еще не загрузила файл COMMAND.COM, который занимается загрузкой EXE-программ в память для преобразования EXE-программы, полученной после работы TLINK, в COM-программу следует использовать утилиту EXE2BIN

- Следует тщательно следить за структурой данных заголовка запроса
  - многих ошибок можно избежать, используя понятие структуры данных (struc)
- В поле связи заголовка устройства должна быть -1 – DOS заменит значение этого поля на соответствующее значение
  - если в этом поле будет не -1, DOS поймет это как наличие второго драйвера. Если на самом деле его нет, возможны неприятности.
- Следует тщательно устанавливать биты атрибутов в заголовке устройства
  - по значению поля атрибутов DOS определяет тип устройства. При неправильной установке битов атрибутов могут не отрабатываться функции, имеющиеся в данном драйвере, а также возможны попытки реагировать на функции, отсутствующие в драйвере.
- Основная процедура должна быть дальней (far)
  - в противном случае возможны неприятности со стеком (со всеми вытекающими из этого последствиями).
- Все переменные должны адресоваться в сегменте кода (CS)
  - по умолчанию транслятор ассемблера относит переменные к сегменту данных (DS). Для отнесения переменных к сегменту кода следует
    - либо использовать префикс (cs:)
    - либо определить значение DS:

```
push cs mov ax,cs
pop ds mov ds,ax
```

- Правильно ли содержимое регистров ES:BX при формировании слова состояния
  - в процессе работы драйвера содержимое этих регистров может быть испорчено.
- Следует следить за тем, чтобы локальные процедуры не портили регистры, используемые драйвером
  - в локальных процедурах используемые регистры лучше сохранит
- Следует сохранять регистры перед вызовом функций BIOS
  - например, прерывание int 10h BIOS портит регистры BP, SI, D
- Следует внимательно следить за соответствием PUSH – POP.

Для отладки можно реализовать каждую команду драйвера в виде отдельной COM-программы. При этом для отладки можно использовать утилиту DEBUG.

Основные трудности возникают при отладке команды инициализации. DOS вызывает драйвер с этой командой сразу после загрузки драйвера. Для ее отладки можно использовать отладочные процедуры.

Организация отдельного стека. Системный стек содержит всего около 20 слов, поэтому особенно на него рассчитывать нельзя. Для организации своего стека следует сделать следующее:

- сохранить SS и SP в переменных
- установить SS и SP на стек внутри драйвера

```
stack_pnt       dw ?   ; Старый указатель стека
stack_seg       dw ?   ; Старый сегмент стека
newstack        db     100h dup(?)  ; 256 байтов нового стека
newstack_top    equ    $-2     ; Вершина нового стека
; Переключение на новый стек
new_stack   proc   near
        cli     ; Запрещение прерываний на всякий случай
        mov    cs:stack_pnt,sp      ; Сохранение старого SP
        mov    cs:stack_seg,ss      ; Сохранение старого SS
        mov    ax,cs                ; Взять текущий сегмент кода
        mov    ss,ax                ; Установить новый сегмент стека
        mov    sp,newstack_top      ; Установить указатель стека
        sti                         ; Разрешение прерывания
        ret
new_stack   endp
; Переключение на старый стек
old_stack    proc   near
        cli
        mov    ss,cs:stack_seg
        mov    sp,cs:stack_pnt
        sti
        ret
old_stack    endp
```

Бит 4 заголовка атрибутов. Этот бит касается драйверов консоли. Он показывает, что драйвер консоли обеспечивает быстрый способ вывода символов. Если этот бит установлен, драйвер должен подготовить вектор прерывания 29h для адресации процедуры быстрого вывода символов (без обработки комбинации Ctrl-C).

Обработчик прерывания 29h нужен только, если установлен бит 4 в слове атрибутов заголовка устройства. Переустановка вектора прерывания 29h добавляется в команду инициализации.

```
; Подпрограмма выполнения быстрого вывода на консоль
int29h:
        sti
        push   ax
        push   bx
        mov    bl,07h          ; Атрибут "белое на черном"
        mov    ah,09h          ; Вывод в режиме телетайпа
```

```
        int     10h
        pop     bx
        pop     ax
        iret
; Инициализация вектора прерывания 29h на int29h
set29h:
        mov     bx0a4h      ; 29h * 4
        lea     ax,int29h   ; Смещение int29h
        mov     [bx],ax     ; Установить смещение вектора
        mov     [bx+2],cs   ; Установоовить сегмент вектора
```

2.8 Вопросы для самопроверки

1. Заголовок драйвера
2. Имя драйвера
3. Процедура стратегия
4. Процедура прерывания
5. Команды драйвера
6. Заголовок запроса драйвера
7. Слово состояния драйвера
8. Атрибуты загружаемого драйвера
9. Обработка команд драйвера
10. Загрузка драйвера
11. Размещение драйвера в памяти ЭВМ
12. Локальные процедуры загружаемого драйвера
13. Выход из драйвера
14. Нормальный выход из драйвера
15. Загружаемый драйвер DOS
16. Выход из драйвера с ошибкой
17. Структура загружаемого драйвера
18. Обработка ошибок в драйвере
19. Трансляция загружаемого драйвера
20. Команда инициализации загружаемого драйвера
21. Компоновка загружаемого драйвера
22. Обращение DOS к драйверу
23. Мультиплексное прерывание 2Fh
24. Проблемы сегмента данных в драйверах
25. Отличие драйвера от резидентной программы
26. Отличие драйвера от исполняемой программы
27. Загружаемый драйвер DOS
28. Структура загружаемого драйвера

29.Трансляция загружаемого драйвера
30.Стандартные команды драйвера
31.Компоновка загружаемого драйвера
32.Структуры данных
33.Заголовок драйвера
34.Сообщение DOS об ошибке драйвера
35.Имя драйвера
36.Символьные и блочные загружаемые драйверы
37.Процедура Стратегия
38.Выход из драйвера с ошибкой
39.Процедура Прерывание
40.Нормальный выход из драйвера
41.Команды драйвера
42.Выход из драйвера
43.Заголовок запроса драйвера
44.Обработка ошибок в драйвере
45.Слово состояния драйвера
46.Команда инициализации загружаемого драйвера
47.Атрибуты драйвера
48.Обращение DOS к драйверу
49.Обработка команд драйвера
50.Выход из драйвера
51.Загрузка драйвера
52.Ошибки в драйвере
53.Главная загрузочная запись
54.Выход из драйвера
55.Команда device=
56.Размещение драйвера в памяти ЭВМ
57.Блок параметров BIOS
*58.*Локальные процедуры драйвера
*59.*Загрузочная

Тема 3 Особенности работы с 32-разрядными процессорами

3.1 Особенности 32-разрядных процессоров

С появлением 32-разрядных процессоров корпорации Intel (80386, i486, Pentium) значительно расширился спектр возможностей программистов. Официально эти процессоры могут работать в трех режимах: реальном,

защищенном и виртуального процессора 8086 (как будет показано ниже, это далеко не все возможные режимы работы) [8].

Каждая следующая модель микропроцессора оказывается значительно совершеннее предыдущей. Так, начиная с процессора i486, арифметический сопроцессор, ранее выступавший в виде отдельной микросхемы, реализуется на одном кристалле с центральным процессором; улучшаются характеристики встроенной кэш-памяти; быстро растет скорость работы процессора. Однако все эти усовершенствования мало отражаются на принципах и методике программирования. Приводимые здесь программы будут одинаково хорошо работать на любом 32-разрядном процессоре. В дальнейшем под термином "процессор" мы будем понимать любую модификацию 32-разрядных процессоров корпорации Intel – от 80386 до Pentium, а также многочисленные разработки других фирм, совместимые с исходными процессорами Intel.

Процессор содержит около 40 программно адресуемых регистров (не считая регистров сопроцессора), из которых шесть являются 16-разрядными, а большая часть остальных – 32-разрядными. Регистры принято объединять в группы: регистры данных, регистры-указатели, сегментные регистры, управляющие регистры, регистры системных адресов, отладочные регистры и регистры тестирования. Кроме того, в отдельную группу выделяют счетчик команд и регистр флагов. На рисунке 7.1 показаны регистры, чаще других используемые в прикладных программах.

Регистры общего назначения и регистры-указатели отличаются от аналогичных регистров процессора 8086 тем, что они являются 32-разрядными.

Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в микропроцессоре 8086/88 (*AX, BX, CX, DX*, *SI, DI, BP* и *SP*). Естественно, сохранена возможность работы с младшими (*AL, BL, CL* и *DL*) и старшими (*AH, BH, CH* и *DH*) половинками регистров МП 8086/88. Однако старшие половины 32-разрядных регистров процессора не имеют мнемонических обозначений и непосредственно недоступны. Для того, чтобы прочитать, например, содержимое старшей половины регистра *EAX* (биты 31...16) придется сдвинуть все содержимое *EAX* на 16 разрядов вправо (в регистр *AX*) и прочитать затем содержимое регистра *AX*.

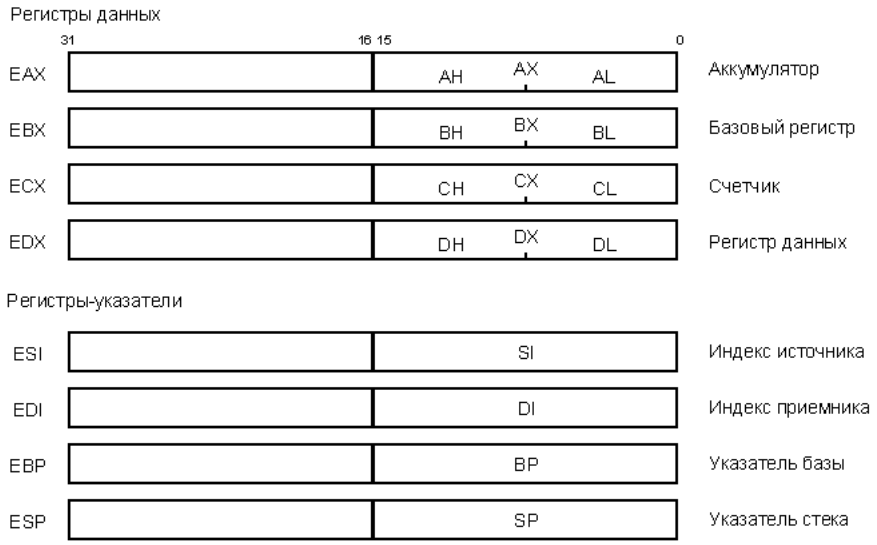


Рисунок 3.1 – Регистры общего назначения

Все регистры общего назначения и указатели программист может использовать по своему усмотрению для временного хранения адресов и данных размером от байта до двойного слова. Так, например, возможно использование следующих команд:

```
mov  EAX,0FFFFFFFFh ; Работа с двойным словом (32 бита)
mov  BX,0FFFFFFFFh  ; Работа со словом (16 бит)
mov  CL,0FFh        ; Работа с байтом (8 бит)
```

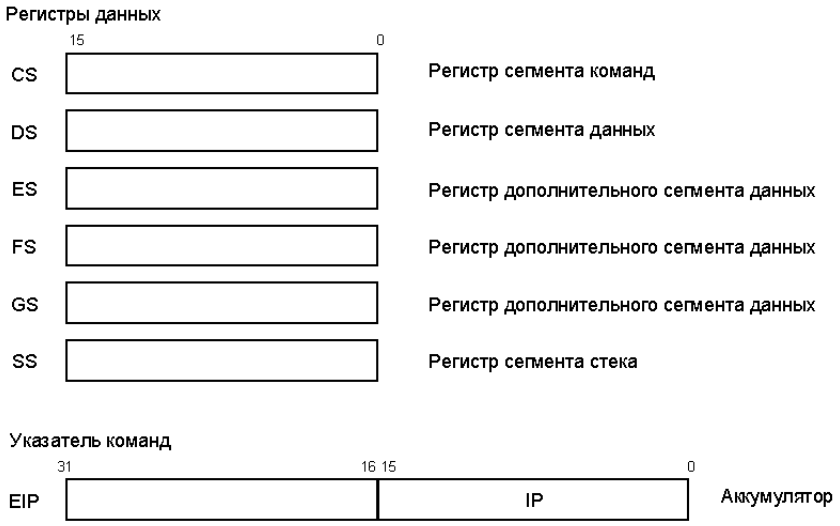


Рисунок 3.2 – Сегментные регистры и указатель команд

Все сегментные регистры, как и в процессоре 8086, являются 16-разрядными. В их состав включено еще два регистра – *FS* и *GS*, которые могут использоваться для хранения сегментных адресов двух дополнительных сегментов данных. Таким образом, при работе в реальном режиме из программы можно обеспечить доступ одновременно к четырем сегментам данных, а не к двум, как при использовании МП 8086.

Регистр указателя команд также является 32-разрядным и обычно при описании процессора его называют *EIP*. Младшие шестнадцать разрядов этого регистра соответствуют регистру *IP* процессора 8086.

Регистр флагов процессоров, начиная с 486 принято называть *EFLAGS*. Дополнительно к шести флагам состояния (*CF, PF, AF, ZF, SF* и *OF*) и трем флагам управления состоянием процессора (*TF, IF* и *DF*), назначение которых было описано в предыдущих пособиях, он включает три новых флага *NT, RF* и *VM* и двухбайтовое поле *IOPL* (рисунок 3.3).

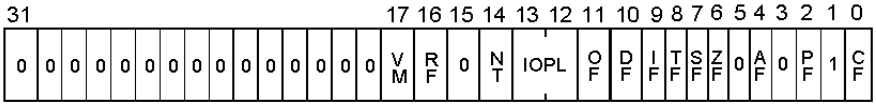


Рисунок 3.3 – Регистр флагов *EFLAGS*

Новые флаги *NT, RF* и *VM*, а также поле *IOPL* используются процессором только в защищенном режиме.

Двухразрядное поле привилегий ввода-вывода *IOPL* (Input/Output Privilege Level) указывает на максимальное значение уровня текущего приоритета (от 0 до 3), при котором команды ввода-вывода выполняются без генерации исключений.

Флаг вложенной задачи *NT* (Nested Task) показывает, является ли текущая задача вложенной в выполнение другой задачи. В этом случае *NT*=1. Флаг устанавливается автоматически при переключении задач. Значение *NT* проверяется командой iret для определения способа возврата в вызвавшую задачу.

Управляющий флаг рестарта *RF* (Restart Flag) используется совместно с отладочными регистрами. Если *RF*=1, то ошибки, возникшие во время отладки при исполнении команды, игнорируются до выполнения следующей команды.

Управляющий флаг виртуального режима *VM* (Virtual Mode) используется для перевода процессора из защищенного режима в режим виртуального процессора 8086. В этом случае процессор функционирует как быстродействующий МП 8086, но реализует механизмы защиты памяти, страничной адресации и ряд других возможностей.

При работе с процессором программист имеет доступ к четырем управляющим регистрам *CR0...CR3*, в которых содержится информация о состоянии компьютера. Эти регистры доступны только в защищенном режиме для программ, имеющих уровень привилегий 0. Нас будет интересовать лишь регистр *CR0* (рисунок 3.4), представляющий собой слово состояния системы. Более подробно этот управляющий регистр будет рассмотрен в следующей главе.

Для управления режимом работы процессора и указания его состояния используются следующие шесть битов регистра *CR0*:

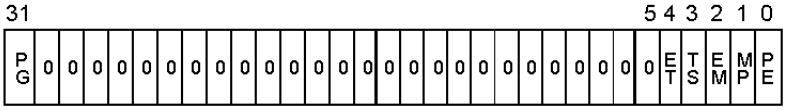


Рисунок 3.4 – Управляющий регистр *CR0*

Бит страничного преобразования *PG* (Paging Enable). Если этот бит установлен, то страничное преобразование разрешено; если сброшен, то запрещено.

Бит типа сопроцессора *ET* (Extension Type) в МП 80286 и 80386 указывал на тип подключенного сопроцессора. Если *ET* = 1, то 80387, если *ET* = 0, то 80287. В более новых процессорах бит *ET* всегда установлен.

Бит переключения задачи *TS* (Task Switched). Этот бит автоматически устанавливается процессором при каждом переключении задачи. Бит может быть очищен командой *clts*, которую можно использоваться только на нулевом уровне привилегий.

Бит эмуляции сопроцессора *EM* (Emulate). Если *EM* = 1, то обработка команд сопроцессора производится программно. При выполнении этих команд или команды *wait* генерируется исключение отсутствия сопроцессора.

Бит присутствия арифметического сопроцессора *MP* (Math Present). Операционная система устанавливает *MP* = 1, если сопроцессор присутствует. Этот бит управляет работой команды *wait*, используемой для синхронизации работы программы и сопроцессора.

Бит разрешения защиты *PE* (Protection Enable). При *PE* = 1 процессор работает в защищенном режиме; при *PE* = 0 в реальном. *PE* может быть установлен при загрузке регистра *CR0* командами *lmsw* или *mov CR0*, а сброшен только командой *mov CR0*.

Регистр *CR1* зарезервирован фирмой Intel для последующих моделей процессоров. Регистры *CR2* и *CR3* служат для поддержки страничного преобразования адреса. Эти два регистра используются вместе. *CR2* содержит полный линейный адрес, вызвавший исключительную ситуацию на последней странице, а *CR3* – адрес, указывающий базу каталога страницы.

Регистры системных адресов (см. рисунок 3.5) используются в защищенном режиме работы процессора. Они задают расположение системных таблиц, служащих для организации сегментной адресации в защищенном режиме.

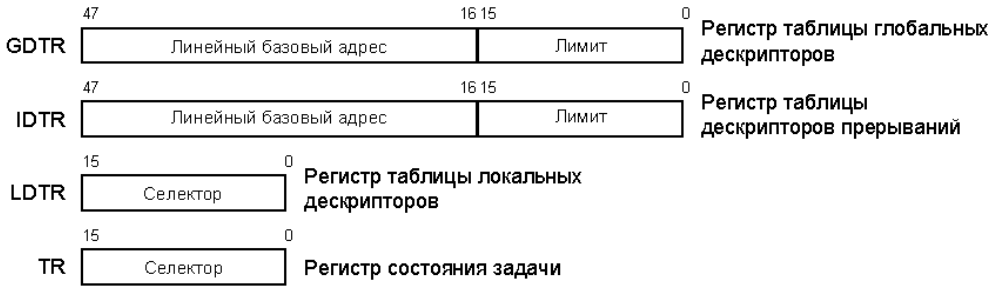


Рисунок 3.5 – Регистры системных адресов

В состав процессора входят четыре регистра системных адресов:

GDTR (Global Descriptor Table Register) – регистр таблицы глобальных дескрипторов для хранения линейного базового адреса и границы таблицы глобальных дескрипторов.

IDTR (Interrupt Descriptor Table Register) – регистр таблицы дескрипторов прерываний для хранения линейного базового адреса и границы таблицы дескрипторов прерываний.

LDTR (Local Descriptor Table Register) – регистр таблицы локальных дескрипторов для хранения селектора сегмента таблицы локальных дескрипторов.

TR (Task Register) – регистр состояния задачи для хранения селектора сегмента состояния задачи.

Рассмотрим листинг простой программы для 32-разрядного процессора.

Листинг 3.1 Программа сложения 32-разрядных операндов

```
.386
Assume CS:code, DS:data, SS:stk

; Простая программа сложения 32-разрядных чисел
data segment para public "data"    ; Сегмент данных
sum  dd      0                     ; Переменная для суммы
data ends

stk  segment para stack "stack"    ; Сегмент стека
     db      256 dup (?)           ; Буфер для стека
stk  ends

code segment para public "code" use16 ; Сегмент кода
begin:
     mov  ax,data    ; Адрес сегмента данных в регистр AX
     mov  ds,ax      ; Запись AX в DS
; Основной фрагмент программы
     mov  eax,12345678h       ; Первый 32-разрядный операнд
     add  eax,87654321h       ; Второй 32-разрадный операнд
```

```
     mov  dword ptr sum,eax   ; Запись результата в sum
; Завершение программы
     mov  ax,4C00h  ; Функция завершения программы
     int  21h       ; Функция Dos
code ends
     END  begin
```

Поскольку в данной программе обрабатываются 32-разрядные числа, в текст программы необходимо включить директиву .386, разрешающую использование команд 32-разрядных процессоров. Кроме того, при компоновке программы с помощью программы **tlink.exe** следует указать ключ /3 для разрешения 32-разрядных операций.

Если рассмотреть листинг этой программы, можно увидеть как команды МП 8086 для работы с 16-разрядными операндами, так и команды МП 386 для работы с 32-разрядными операндами. Для облегчения текста из протокола трансляции удалены строчные комментарии.

```
1    .386
2    Assume    CS:code, DS:data, SS:stk
3
4    ; Простая программа сложения 32-разрядных чисел
5    00000000  data segment para public "data"
6    00000000  00000000  sum   dd   0
7    00000004  data      ends
8
9    00000000  stk  segment para stack "stack"
10   00000000  0100*(??) db    256 dup (?)
11   00000100  stk  ends
12
13   0000      code segment para public "code" use16
14   0000      begin:
15   0000 B8   0000s     mov   ax,data
16   0003 8E   D8        mov   ds,ax
17   ; Основной фрагмент программы
18   0005 66| B8 12345678      mov eax,12345678h
19   000B 66| 05 87654321      add eax,87654321h
20   0011 66| 67| A3 00000000r mov  dword ptr sum,eax
21   ; Завершение программы
22   0018 B8   4C00      mov   ax,4C00h
23   001B CD   21        int   21h
24   001D code ends
25        END  begin
```

В строках 15 и 16 листинга используется команда засылки операнда в аккумулятор (B8h). Однако в строке 18 наличие перед кодом этой команды префикса замены размера операнда (код 66h) определяет, что длина операнда равна 32 бита, и, следовательно, используется регистр *EAX*. Префикс замены размера операнда включается в объектный модуль транслятором автоматически, если в программе указано мнемоническое обозначение 32-разрядного регистра, например, *EAX*.

При отладке этой программы используется отладчик фирмы Borland (турбо дебаггер).

Для индикации содержимого 32-разрядных регистров требуется провести дополнительную настройку отладчика. Запустив отладчик, надо выбрать *Основное меню→View→Registers*. При этом откроется окно индикации содержимого регистров процессора. Затем необходимо вызвать локальное меню этого окна, нажав *ALT-F10*, выбрать в открывшемся меню пункт *Registers 32 bit* и нажать *Enter*. После этого стоявшее по умолчанию в этом пункте *No* сменится на *Yes*. Это обеспечит вывод на экран содержимого полных 32-разрядных регистров *EAX...ESP* взамен 16-разрядных регистров *AX...SP*. Окно отладчика с исходным состоянием программы и переменных показано на рисунке 3.6.

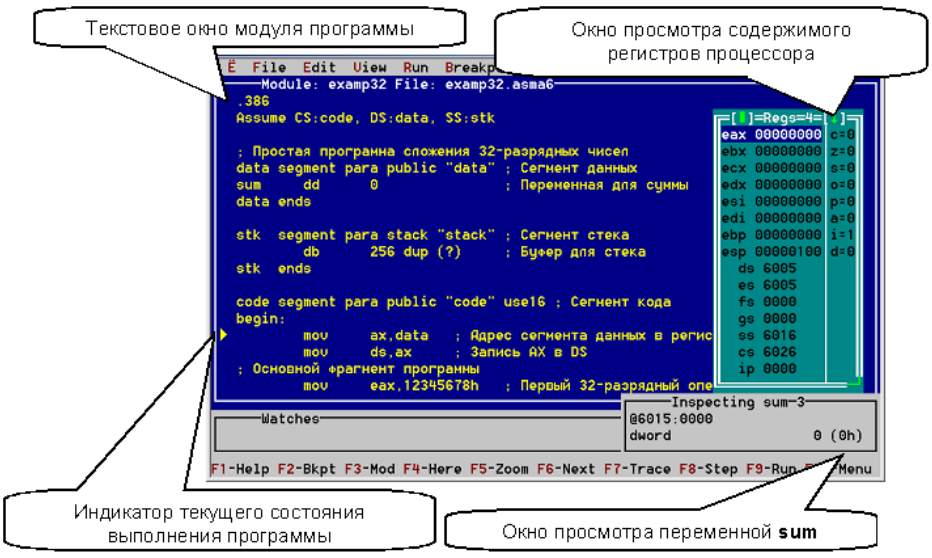


Рисунок 3.6 – Окно отладчика с исходным состоянием программы

Для иллюстрации выполнения 32-разрядного сложения надо выполнить программу до команды пересылки содержимого *EAX* в переменную ***sum*** включительно (строка программы 20). Для этого следует 5 раз нажать клавишу F7, которая вызывает покомандное выполнение программы. Результат такого выполнения показан на рисунке 7.7.

На рисунке 3.6 видно, что содержимое аккумулятора в окне просмотра регистров процессора и содержимое переменной ***sum*** в окне просмотре переменных нулевые. На рисунке 3.7 содержимое и аккумулятора и указанной переменной уже равно 99999999h (или 2576980377 десятичных), что является результатом сложения 12345678h и 87654321h.

Кроме значения рассматриваемой переменной в окне просмотра переменных указан еще тип переменной (dword) и ее адрес (6015:0000).

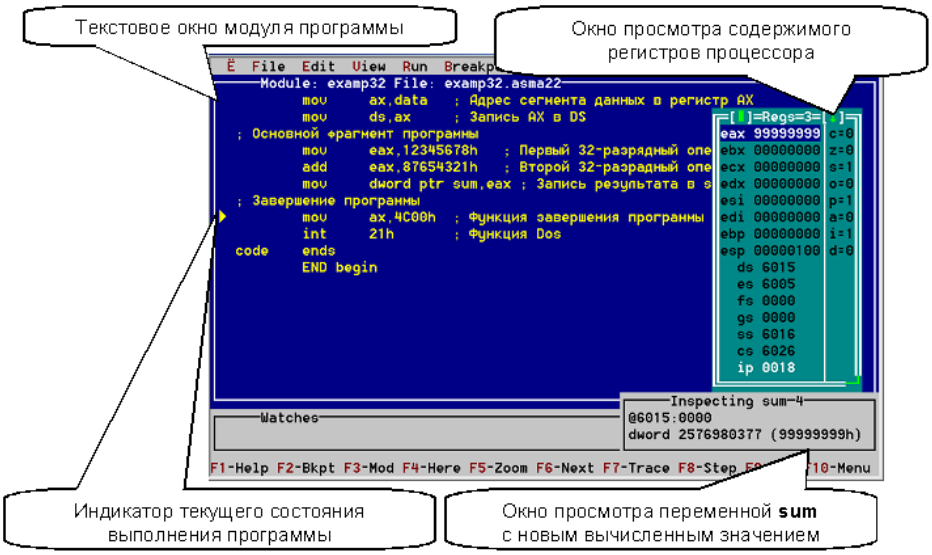


Рисунок 3.7 – Окно отладчика с результатом выполнения 32-разрядного сложения

В рассмотренном листинге используются уже известные нам команды. Однако в систему команд современных процессоров включен ряд новых команд, выполнение которых не поддерживается процессором 8086. Некоторые из этих команд впервые появились в процессоре 80386, другие – в процессорах i486 или Pentium. Ниже приведен список этих команд.

Команды общего назначения

bound – проверка индекса массива относительно границ массива.

bsf/bsr – команды сканирования битов.

bt/btc/btr/bts – команды выполнения битовых операций.

bswap – изменение порядка байтов операнда.

cdq – преобразование двойного слова в четверное.

cmpsd – сравнение строк по двойным словам.

cmpxchg – сравнение и обмен операндов.

cmpxchg8b – сравнение и обмен 8-битовых операндов.

cpuid – идентификация процессора

cwde – преобразование слова в двойное слово с расширением.

enter – создание кадра стека для параметров процедур.

imul *reg,imm* – умножение операнда со знаком на непосредственное значение.

ins/outs – ввод/вывод из порта в строку.

iretd – возврат из прерывания в 32-разрядном режиме.

j(cc) – команды условного перехода, допускающие 32-битовое смещение.

leave – выход из процедуры с удалением кадра стека, созданного командой enter.

lss/lfs/lgs – команды загрузки сегментных регистров.

mov *DRx,reg; reg,DRx*

mov *CRx,reg; reg,CRx*

mov *TRx,reg; reg,TRx* – команды обмена данными со специальными регистрами. В качестве источника или приемника могут быть использованы регистры *CR0...CR3, DR0...DR7, TR3...TR5*.

movsx/movzx – знаковое/беззнаковое расширение до размера приемника и пересылка.

popa – извлечение из стека всех 16-разрядных регистров общего назначения (*AX, BX, CX, DX, SP, BP, SI, DI*).

popad – извлечение из стека всех 32-разрядных регистров общего назначения (*EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI*).

push *imm* – запись в стек непосредственного операнда размером байт, слово или двойное слово (например, *push* OFFFFFFFFh).

pusha – запись в стек всех 16-разрядных регистров общего назначения (*AX, BX, CX, DX, SP, BP, SI, DI*).

pushad – запись в стек всех 32-разрядных регистров общего назначения (*EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI*).

rcr/rcl/ror/rol *reg/mem,imm* – циклический сдвиг на непосредственное значение.

sar/sal/shr/shl *reg/mem,imm* – арифметический сдвиг на непосредственное значение.

scasd – сканирование строки двойных слов с целью сравнения.

set(cc) – установка байта по условию.

shrd/shld – логический сдвиг с двойной точностью.

stosd – запись двойного слова с строку.

xadd – обмен и сложение.

xlatb – табличная трансляция.

Команды защищенного режима

arpl – корректировка поля *RPL* селектора

clts – сброс флага переключения задач в регистре *CR0*.

lar – загрузка байта разрешения доступа.
lgdt – загрузка регистра таблицы глобальных дескрипторов.
lidt – загрузка регистра таблицы дескрипторов прерываний.
lldt – загрузка регистра таблицы локальных дескрипторов.
lmsw – загрузка слова состояния машины.
lsl – загрузка границы сегмента.
ltr – загрузка регистра задачи.
rdmsr – чтение особого регистра модели.
sgdt – сохранение регистра таблицы глобальных дескрипторов.
sidt – сохранение регистра таблицы дескрипторов прерываний.
sldt – сохранение регистра таблицы локальных дескрипторов.
smsw – сохранение слова состояния.
ssl – сохранение границы сегмента
str – сохранение регистра задачи.
verr – проверка доступности сегмента для чтения.
verw – проверка доступности сегмента для записи.

3.2 Первое знакомство с защищенным режимом

Как уже отмечалось, современные процессоры могут работать в трех режимах: реальном, защищенном и виртуального 86-го процессора. В реальном режиме процессоры функционируют фактически так же, как МП 8086 с повышенным быстродействием и расширенным набором команд. Многие весьма привлекательные возможности процессоров принципиально не реализуются в реальном режиме, который введен лишь для обеспечения совместимости с предыдущими моделями. Все программы, приведенные в предыдущих пособиях по системному программному обеспечению, относятся к реальному режиму и могут с равным успехом выполняться на любом из этих процессоров без каких-либо изменений. Характерной особенностью реального режима является ограничение объема адресуемой оперативной памяти величиной 1 Мбайт.

Только перевод микропроцессора в защищенный режим позволяет полностью реализовать все возможности, заложенные в его архитектуру и недоступные в реальном режиме. Сюда можно отнести:

- увеличение адресуемого пространства до 4 Гбайт;
- возможность работы в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти и достигающей огромной величины 64 Тбайт. Правда, для реализации виртуального режима необходимы, помимо дисков большой емкости, еще и соответствующая операционная система,

которая хранит все сегменты выполняемых программ в большом дисковом пространстве, автоматически загружая в оперативную память те или иные сегменты по мере необходимости;

- организация многозадачного режима с параллельным выполнением нескольких программ (процессов). Собственно говоря, многозадачный режим организует многозадачная операционная система, однако микропроцессор предоставляет необходимый для этого режима мощный и надежный механизм защиты задач друг от друга с помощью четырехуровневой системы привилегий;
- страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

При включении процессора в нем автоматически устанавливается реальный режим. Переход в защищенный режим осуществляется программно путем выполнения соответствующей последовательности команд. Поскольку многие детали функционирования процессора в реальном и защищенном режимах существенно различаются, программы, предназначенные для защищенного режима, должны быть написаны особым образом. Реальный и защищенный режимы не совместимы! Архитектура современного микропроцессора необычайно сложна. Столь же сложными оказываются и программы, использующие средства защищенного режима. К счастью, однако, отдельные архитектурные особенности защищенного режима оказываются в достаточной степени замкнутыми и не зависящими друг от друга. Так, при работе в однозадачном режиме отпадает необходимость в изучении многообразных и замысловатых методов взаимодействия задач. Во многих случаях можно отключить (или, точнее, не включать) механизм страничной организации памяти. Часто нет необходимости использовать уровни привилегий. Все эти ограничения существенно упрощают освоение защищенного режима

Начнем изучение защищенного режима с рассмотрения простейшей (но, к сожалению, все же весьма сложной) программы, которая, будучи запущена обычным образом под управлением MS-DOS, переключает процессор в защищенный режим, выводит на экран для контроля несколько символов, переходит назад в реальный режим и завершается стандартным для DOS образом [8]. Рассматривая эту программу, мы познакомимся с основополагающей особенностью защищенного режима – сегментной адресацией памяти, которая осуществляется совсем не так, как в реальном режиме.

Следует заметить, что для выполнения рассмотренной ниже программы необходимо, чтобы на компьютере была установлена система

MS-DOS "в чистом виде" (не в виде сеанса DOS системы Windows). Перед запуском программ защищенного режима следует выгрузить как систему Windows, так и драйверы обслуживания расширенной памяти HIMEM.SYS и EMM386.EXE.

Программа, работающая в защищенном режиме

```
;*********************************************
; Программа, работающая в защищенном режиме  *
;*********************************************
   .386P ; Разрешение трансляции всех (в том
         ; числе и привилегированных команд
         ; процессоров 386 и 486
; Структура для дескриптора сегмента
descr   struc
lim     dw      0 ; Граница (биты 0 - 15)
base_1  dw      0 ; База (биты 0 - 15)
base_m  db      0 ; База (биты 16 - 23)
attr_1  db      0 ; Байт атрибутов 1
attr_2  db      0 ; Граница (биты 16 - 19)
                  ; и атрибуты 2
base_h  db      0 ; База (биты 24 - 31)
descr   ends

data    segment  use16
;  Таблица глобальных дескрипторов GDT
; Селектор 0 – обязательный нулевой дескриптор
gdt_null  descr <0,0,0,0,0,0>
; Селектор 8 - сегмент данных
gdt_data  descr <data_size-1,0,0,92h,0,0>
; Селектор 16 - сегмент кода
gdt_code  descr <code_size-1,0,0,98h,0,0>
; Селектор 24 – сегмент стека
gdt_stack descr <255,0,0,92h,0,0>
;Селектор 32 - видеобуфер
gdt_screen descr <4095,8000h,0bh,92h,0,0>
gdt_size=$-gdt_null      ; Размер GDT
;=================================================
; Поля данных программы
pdescr  dq    0     ; Псевдодескриптор для команды lgdt
sym     db    1     ; Символ для вывода на экран
attr    db    1ah   ; Атрибут символа
mes     db    27,'[31;42mReal mode now',27,'[0m',10,13,'$'
mes1    db    26 dup(32),'A message in protected mode',27 dup(32),0
data_size=$-gdt_null           ; Размер сегмента данных
data    ends
;=================================================
```

```
41 text    segment    'code'  use16 ; По умолчанию 16-разрядный режим
42         assume    cs:text,ds:data
43 main    proc
44      xor eax,eax  ; Очистка 32-разр. EAX
45      mov ax,data  ; Инициализация сегментного регистра
46      mov ds,ax    ; для реального режима
47 ; Вычисление 32-битного линейного адреса сегмента данных и загрузка
48 ; его в дескриптор (в EAX уже находится его сегментный адрес).
49 ; Для умножения его на 16 сдвинем его влево на 4 разряда
50      shl eax,4       ; В EAX - линейный базовый адрес
51      mov ebp,eax     ; Сохранение его в EBP
52      mov bx,offset gdt_data     ; В BX адрес дескриптора
53      mov [bx].base_1,ax    ; Мл. часть базы
54      rol eax,16      ; Обмен старшей и младшей половины EAX
55      mov [bx].base_m,al    ; Средняя часть базы
56 ;Вычисление и загрузка 32-битного линейного адреса сегмента команд
57      xor eax,eax     ; Очистка 32-разр. EAX
58      mov ax,cs       ; Адрес сегмента команд
59      shl eax,4       ; В EAX - линейный базовый адрес
60      mov bx,offset gdt_code     ; В BX адрес дескриптора
61      mov [bx].base_1,ax    ; Мл. часть базы
62      rol eax,16      ; Обмен старшей и младшей половины EAX
63      mov [bx].base_m,al    ; Средняя часть базы
64 ;Вычисление и загрузка 32-битного линейного адреса сегмента стека
65      xor eax,eax
66      mov ax,ss
67      shl eax,4
68      mov bx,offset gdt_stack
69      mov [bx].base_1,ax
70      rol eax,16
71      mov [bx].base_m,al
72 ;Подготовка псевдодескриптора и загрузка его в регистр GDTR
73      mov dword ptr pdescr+2,ebp             ; База GDT (0-31)
74      mov word ptr pdescr,gdt_size-1   ; Граница GDT
75      lgdt    pdescr                   ; Загрузка регистра GDTR
76 ;Подготовка к переходу в защищенный режим
77      cli             ; Запрет маскир. прерываний
78      mov al,80h      ; Запрет NMI
79      out 70h,al      ; Порт КМОП микросхемы
80 ;Переход в защищенный режим
81      mov eax,cr0  ; Чтение регистра состояния
82      or  eax,1    ; Взведение бита 0
83      mov cr0,eax
84 ;************************************************
85 ;* Теперь процессор работает в защищенным режиме *
86 ;************************************************
```

```
87 ; Загрузка в CS селектор сегмента кода, а в IP смещения следующей
88 ; команды (при этом и очищается очередь команд)
89      db  0eah              ; Код команды far jmp
90      dw  offset continue   ; Смещение
91      dw  16                ; Селектор сегмента команд
92 continue:
93 ;Инициализация селектора сегмента данных
94      mov ax,8        ; Селектор сегмента данных
95      mov ds,ax
96 ;Инициализация селектора сегмента стека
97      mov ax,24       ; Селектор сегмента стека
98      mov ss,ax
99 ;Инициализация селектора ES и вывод символов на экран
100     mov ax,32       ; Селектор сегмента видеобуфера
101     mov es,ax
102     mov ebx,800     ; Начальное смещение на экране
103;Вывод сообщения на экран
104     lea esi,mes1
105     mov ah,attr
106screen:
107     mov al,[esi]
108     or  al,al
109     jz  scend ; Выход, если нуль (терминатор сообщения)
110     mov es:[bx],ax  ; Вывод символа в видеобуфер
111     add ebx,2       ; Следующий адрес на экране
112     inc esi         ; Следующий символ
113     jmp screen      ; Цикл
114 scend:                          ; Конец вывода сообщения
115 ;Подготовка перехода в реальный режим
116 ;************************************************
117 ;Формирование и загрузка дескрипторов для реального режима
118     mov gdt_data.lim,0ffffh    ; Запись значения
119     mov gdt_code.lim,0ffffh    ; границы в 4 ис-
120     mov gdt_stack.lim,0ffffh   ; пользуемых нами
121     mov gdt_screen.lim,0ffffh ; дескриптора
122 ; Для перенесения этих значений в теневые регистры необходимо
123 ; записать в сегментные регистры соответствующие селекторы
124     mov ax,8  ; Загрузка теневого регистра
125     mov ds,ax ; сегмента данных
126     mov ax,24 ; Загрузка теневого регистра
127     mov ss,ax ; сегмента стека
128     mov ax,32 ; Загрузка теневого регистра
129     mov es,ax ; дополнительного сегмента
130 ;Сегментный регистр CS программно недоступен, поэтому его
131 ;   загрузку опять выполняем косвенно с помощью искусственно
132 ;   сформированной команды дальнего перехода
```

```
133     db  0eah       ; Код команды дальнего перехода
134     dw  offset go ; Смещение
135     dw  16        ; Селектор сегмента кода
136 ;Переключение режима процессора
137 go: mov eax,cr0          ; Чтение cr0
138     and eax,0fffffffeh          ; Сброс бита 0
139     mov cr0,eax           ; Запись cr0
140     db  0eah              ; Код команды дальнего перехода
141     dw  return            ; Смещение
142     dw  text              ; Сегмент кода
143 ;*******************************************************
144 ;* Теперь процессор опять работает в реальном режиме *
145 ;*******************************************************
146 ;Восстановление операционной среды реального режима
147 return:
148     mov ax,data ; Инициализация сегментных регистров
149     mov ds,ax   ; данных и
150     mov ax,stk  ; стека
151     mov ss,ax   ; в реальном режиме
152 ;Мы не восстанавливает содержимое SP, так как при таком мягком (без
153 ;   сброса) переходе в реальный режим SP не разрушается
154 ;Разрешение всех прерываний
155     sti             ; Разрешение маск. прерываний
156     mov al,0        ; Сброс бита 7 порта 70 КМОП -
157     out 70h,al      ; разрешение NMI
158 ;Вывод сообщения в реальном режиме
159     mov ah,9         ; Вывод сообщения
160     mov dx,offset mes  ; функцией DOS
161     int 21h
162 ;Ожидание нажатия клавиши
163     xor ah,ah
164     int 16h
165 ; Завершение программы
166     mov ax,4c00h
167     int 21h
168 main   endp
169 code_size=$-main
170 text   ends
171 stk    segment    stack 'stack'
172        db         256 dup(0)
173 stk    ends
174        end        main
```

К тексту программы добавлены номера строк, которые облегчат описание отдельных команд в тексте. Следует иметь в виду, что перед

трансляцией показанного текста, необходимо удалить все номера строк, так как компилятор на каждый номер будет выдавать ошибку.

32-разрядные микропроцессоры отличаются расширенным набором команд, часть которых относится к привилегированным. Для того, чтобы разрешить транслятору обрабатывать эти команды, в текст программы необходимо включить директиву ассемблера .386P.

Программа начинается с описания структуры дескриптора сегмента. В отличие от реального режима, в котором сегменты определяются их базовыми адресами, задаваемыми программистом в явной форме, в защищенном режиме для каждого сегмента программы должен быть определен дескриптор – 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики (рисунок 3.8) [8].

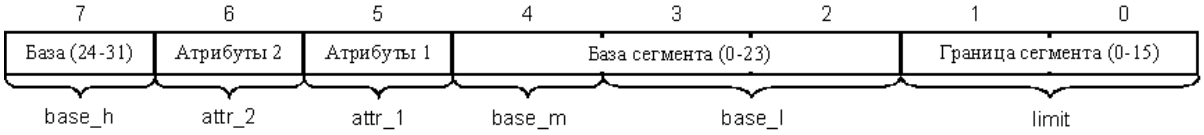


Рисунок 3.8 – Дескриптор сегмента

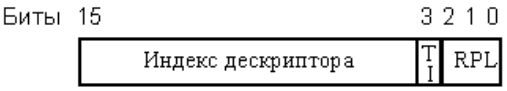


Рисунок 3.9 – Селектор дескриптора

Теперь для обращения к требуемому сегменту программист заносит в сегментный регистр не сегментный адрес, а так называемый селектор (рисунок 7.9), в состав которого входит номер (индекс) соответствующего сегмента дескриптора.

Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя к нему указанное в конкретной команде смещение (относительный адрес), формирует адрес ячейки памяти. Индекс дескриптора (0, 1, 2 и т.д.) записывается в селектор, начиная с бита 3, что эквивалентно умножению его на 8. Таким образом, можно считать, что селекторы последовательных дескрипторов представляют собой числа 0, 8, 16, 24 и т.д. Другие поля селектора, которые для нашего случая принимают значения 0, будут описаны ниже.

Структура *descr* (строка 8 листинга 1.1) предоставляет шаблон для дескрипторов сегментов, облегчающий их формирование. Сравнивая описание структуры *descr* в программе с рисунком 1.8, нетрудно заметить их соответствие друг другу.

Рассмотрим вкратце содержимое дескриптора. Граница (*limit*) сегмента представляет собой номер последнего байта сегмента. Так, для сегмента размером 375 байт граница равна 374. Поле границы состоит из 20 бит и разбито на две части. Как видно из рисунка 7.8, младшие 16 бит границы занимают байты 0 и 1 дескриптора, а старшие 4 бита входят в байт атрибутов 2, занимая в нем биты 0...3. Получается, что размер сегмента ограничен величиной 1 Мбайт. На самом деле это не так. Граница может указываться либо в байтах (и тогда, действительно, максимальный размер сегмента равен 1 Мбайт), либо в блоках по 4 Кбайт (и тогда размер сегмента может достигать 4 Гбайт). В каких единицах задается граница, определяет старший бит байта атрибутов 2, называемый битом дробности. Если он равен 0, граница указывается в байтах; если 1 – в блоках по 4 килобайта.

База сегмента (32 бита) определяет начальный линейный адрес сегмента в адресном пространстве процессора. Линейным называется адрес, выраженный не в виде комбинации сегмент-смещение, а просто номером байта в адресном пространстве. Казалось бы, линейный адрес – это просто другое название физического адреса. Для нашей программы это так и есть, в нем линейные адреса совпадают с физическими. Однако если в процессоре включен блок страничной организация памяти, то процедура преобразования адресов усложняется. Отдельные блоки размером 4 Кбайт (страницы) линейного адресного пространства могут произвольным образом отображаться на физические адреса, в частности и так, что большие линейные адреса отображаются на начало физической памяти, и наоборот. Страничная адресация осуществляется аппаратно (хотя для ее включения требуются определенные программные усилия) и действует независимо от сегментной организации программы. Поэтому во всех программных структурах защищенного режима фигурируют не физические, а линейные адреса. Если страничная адресация выключена, эти линейные адреса совпадают с физическими, если включена – могут и не совпадать.

Страничная организация повышает эффективность использования памяти программами, однако практически она имеет смысл лишь при выполнении больших по размеру задач, когда объем адресного пространства задачи (виртуального адресного пространства) превышает наличный объем памяти. В рассмотренном листинге используется чисто сегментная адресация без деления на страницы, и линейные адреса совпадают с физическими.

Поскольку в дескриптор записывается 32-битовый линейный базовый адрес (номер байта), сегмент в защищенном режиме может начинаться на любом байте, а не только на границе параграфа, и располагаться в любом месте адресного пространства 4 Гбайт.

Поле базы, как и поле границы, разбито на 2 части: биты 0...23 занимают байты 2, 3 и 4 дескриптора, а биты 24...31 – байт 7. Для удобства программного обращения в структуре *descr* база описывается тремя полями: младшим словом (base_l – строка 10 листинга) и двумя байтами: средним (base_m – строка 11 листинга) и старшим (base_h – строка 15 листинга).

В байте атрибутов 1 задается ряд характеристик сегмента. Не вдаваясь пока в подробности этих характеристик, укажем, что в рассмотренной программе используются сегменты двух типов: сегмент команд, для которого байт *attr_1* (строка 12 листинга) должен иметь значение 98h, и сегмент данных (или стека) с кодом 92h.

Некоторые дополнительные характеристики сегмента указываются в старшем полубайте байта *attr_2* (в частности, бит дробности). Для всех наших сегментов значение этого полубайта равно 0.

Сегмент данных *data* (строка 18 листинга), который для удобства изучения функционирования программы расположен в начале программы, до сегмента команд, объявлен с типом использования *use16* (так же будет объявлен и сегмент команд). Этот описатель объявляет, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса. Если бы мы готовили нашу программу для работы под управлением операционной системы защищенного режима, реализующей все возможности микропроцессора, тип использования был бы *use32*. Однако наша программа будет запускаться под управлением DOS, которая работает в реальном режиме с 16-битовыми адресами и операндами.

Сегмент данных начинается с описания важнейшей системной структуры – таблицы глобальных дескрипторов. Как уже отмечалось выше, обращение к сегментам в защищенном режиме возможно только через дескрипторы этих сегментов. Таким образом, в таблице дескрипторов должно быть описано столько дескрипторов, сколько сегментов использует программа. В нашем случае в таблицу включены, помимо обязательного нулевого дескриптора, всегда занимающего первое место в таблице, четыре дескриптора для сегментов данных, команд, стека и дополнительного сегмента данных, который мы наложим на видеобуфер, чтобы обеспечить возможность вывода в него символов. Порядок дескрипторов в таблице (кроме нулевого) не имеет значения.

Помимо единственной таблицы глобальных дескрипторов, обозначаемой *GDT* от Global Descriptor Table, в памяти может находиться множество таблиц локальных дескрипторов (*LDT* от Local Descriptor Table). Разница между ними в том, что сегменты, описываемые глобальными дескрипторами, доступны всем задачам, выполняемым процессором, а к

сегментам, описываемым локальными дескрипторами, может обращаться только та задача, в которой эти дескрипторы описаны. Поскольку пока мы имеем дело с однозадачным режимом, локальная таблица нам не нужна.

Поля дескрипторов для наглядности заполнены конкретными данными явным образом, хотя объявление структуры *descr* с нулями во всех полях позволяет описать дескрипторы несколько короче [8], например:

```
gdt_null descr<>; Селектор 0 – обязательный нулевой дескриптор
gdt_data descr<data_size-1,,,92h> ; Селектор 8 – сегмент данных
```

В дескрипторе *gdt_data* (строка 23 листинга), описывающем сегмент данных программы, заполняется поле границы сегмента (фактическое значение размера сегмента *data_size* будет вычислено компилятором, строка 30 листинга), а также байт атрибутов 1. Код 92h говорит о том, что это сегмент данных с разрешением записи и чтения. Базу сегмента, т.е. физический адрес его начала, придется вычислить программно и занести в дескриптор уже на этапе выполнения.

Дескриптор *gdt_code* (строка 25 листинга) сегмента команд заполняется схожим образом. Код атрибута 98h обозначает, что это исполняемый сегмент, к которому, между прочим, запрещено обращение с целью чтения или записи. Таким образом, сегменты команд в защищенном режиме нельзя модифицировать по ходу выполнения программы.

Дескриптор *gdt_stack* (строка 27 листинга) сегмента стека имеет, как и любой сегмент данных, код атрибута 92h, что разрешает его чтение и запись, и явным образом заданную границу 255 байтов, что соответствует размеру стека. Базовый адрес сегмента стека так же будет вычислен на этапе выполнения программы.

Последний дескриптор *gdt_screen* (строка 29 листинга) описывает страницу 0 видеобуфера. Размер видеостраницы, как известно, составляет 4096 байтов, поэтому в поле границы указано число 4095. Базовый физический адрес страницы известен, он равен B8000h. Младшие 16 разрядов базы (число 8000h) заполняют слово base_l дескриптора, биты 16... 19 (число 0bh) – байт *base_m*. Биты 20...31 базового адреса равны 0, поскольку видеобуфер размещается в первом мегабайте адресного пространства.

Перед переходом в защищенный режим процессору надо будет сообщить физический адрес таблицы глобальных дескрипторов и ее размер (точнее, границу). Размер GDT определяется на этапе трансляции в строке 30.

Назначение оставшихся строк сегмента данных станет ясным в процессе рассмотрения программы.

Сегмент команд *text* (строка 41 листинга) начинается, как и всегда, оператором *segment*, в котором указывается тип использования *use16*, так как мы составляем 16-разрядное приложение. Указание описателя *usel6*.Не запрещает использовать в программе 32-битовые регистры.

Фактически весь листинг программы, кроме ее завершающих строк, а также фрагмента, выполняемого в защищенном режиме, посвящена подготовке перехода в защищенный режим. Прежде всего, надо завершить формирование дескрипторов сегментов программы, в которых остались незаполненными базовые адреса сегментов. Базовые (32-битовые) адреса определяются путем умножения значений сегментных адресов на 16. Сначала производится очистка 32-разрядного аккумулятора (строка 44), так как в нем будет сформирован позднее базовый 32-разрядный адрес (фактически эта команда нужна для очистки старшего слова расширенного аккумулятора). После обычной инициализации сегментного регистра *DS* (строки 45, 46), которая позволит нам обращаться к полям данных программы (в реальном режиме!) выполняется сдвиг на 4 разряда содержимого регистра *EAX*. Эта операция выполняется командой *shl EAX,4.* Команда сдвигает влево содержимое 32-разрядного аккумулятора на указанное константой число бит (4). Следующая команда сохраняет получившееся 32-разрядное значение адреса в регистре *EBP*. После этого в регистр *BX* помещается смещение дескриптора сегмента кода. Следующими тремя командами (строки 53 – 55) младшее и старшее слова регистра *EAX* отправляется в поля *base_l* и *base_m* дескриптора *gdt_data*, соответственно. Аналогично вычисляются 32-битовые адреса сегментов команд и стека, помещаемые в дескрипторы *gdt_code* (строки 57 – 63) и *gdt_stack* (строки 65 – 71).

Следующий этап подготовки к переходу в защищенный режим – загрузка в регистр процессора *GDTR* (Global Descriptor Table Register, регистр таблицы глобальных дескрипторов) информации о таблице глобальных дескрипторов. Эта информация включает в себя линейный базовый адрес таблицы и ее границу и размещается в 6 байтах поля данных, называемого псевдодескриптором. Для загрузки *GDTR* предусмотрена специальная привилегированная команда *lgdt* (load global descriptor table, загрузка таблицы глобальных дескрипторов), которая требует указания в качестве операнда имени псевдодескриптора. Формат псевдодескриптора приведен на рисунке 3.10.

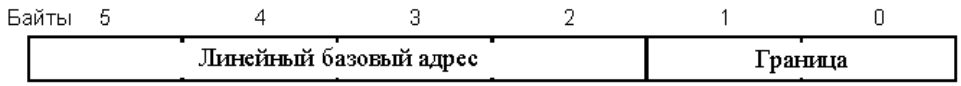


Рисунок 3.10 – Формат псевдодескриптора

В нашем листинге заполнение псевдодескриптора упрощается вследствие того, что таблица глобальных дескрипторов расположена в начале сегмента данных, и ее базовый адрес совпадает с базовым адресом всего сегмента, который уже был вычислен и помещен в дескриптор *gdt_data*. В строках 73, 74 базовый адрес и граница помещаются в требуемые поля *pdescr*, а в строке 75 командой *lgdt* загружается регистр *GDTR,* сообщая, таким образом, процессору о местонахождение и размер *GDT*.

В принципе теперь можно перейти в защищенный режим. Однако мы запускаем нашу программу под управлением DOS (а как ее еще можно запустить?) и естественно завершить ее также обычным образом, чтобы не нарушить работоспособность системы. Но в защищенном режиме запрещены любые обращения к функциям DOS или BIOS. Причина этого совершенно очевидна – и DOS, и BIOS являются программами реального режима, в которых широко используется сегментная адресация реального режима, т.е. загрузка в сегментные регистры сегментных адресов. В защищенном же режиме в сегментные регистры загружаются не сегментные адреса, а селекторы. Кроме того, обращение к функциям DOS и BIOS осуществляется с помощью команд *int* с определенными номерами, а в защищенном режиме эти команды приведут к совершенно другим результатам. Следовательно, программу, работающую в защищенном режиме, нельзя завершить средствами DOS. Сначала ее надо вернуть в реальный режим.

Возврат в реальный режим можно осуществить сбросом процессора. Действия процессора после сброса определяются одной из ячеек КМОП-микросхемы – байтом состояния отключения, располагаемым по адресу Fh. В частности, если в этом байте записан код Ah, после сброса управление немедленно передается по адресу, который извлекается из двухсловной ячейки 40h:67h, расположенной в области данных BIOS. Таким образом, для подготовки возврата в реальный режим необходимо в ячейку 40h:67h записать адрес возврата, а в байт Fh КМОП-микросхемы занести код Ah. Приведенный способ возврата в реальный режим использовался в процессорах i286 (так как другого способа возврата в нем предусмотрено не было).

В процессорах, начиная с i386, переход из реального режима в защищенный и обратно может осуществляться с использованием

управляющего регистра *CR0*. Так как встретить сейчас «живой» 286 процессор практически не реально, воспользуемся именно таким способом.

Всего в микропроцессорах i386 и выше имеется 4 программно адресуемых управляющих регистра *CR0, CR1, CR2* и *CR3* [8]. Регистр *CR1* зарезервирован, регистры *CR2* и *CR3* управляют страничным преобразованием, которое в рассматриваемой программе не используется, а регистр *CR0* содержит набор управляющих битов, из которых нам интересны биты 0 (включение и выключение защищенного режима) и 31 (разрешение страничного преобразования).

После сброса процессора оба эти бита сброшены, благодаря чему процессор начинает работать в реальном режиме с выключенным страничным преобразованием. Установка младшего бита *CR0* в 1 переводит процессор в защищенный режим, а сброс его возвращает процессор в реальный режим. Следует отметить, что младшая половина регистра *CR0* совпадает со словом состояния 286 процессора, поэтому команды чтения и записи в регистр *CR0* аналогичны по результату командам *smsw* и *lmsw*, которые сохранены в старших процессорах из соображений совместимости.

Еще один важный шаг, который необходимо выполнить перед переходом в защищенный режим, заключается в запрете всех аппаратных прерываний. Дело в том, что в защищенном режиме процессор выполняет процедуру обработки прерывания иначе, чем в реальном. При поступлении сигнала прерывания процессор не обращается к таблице векторов прерываний в первом килобайте памяти, как в реальном режиме, а извлекает адрес программы обработки прерывания из таблицы дескрипторов прерываний, построенной аналогично таблице глобальных дескрипторов и располагаемой в программе пользователя (или в операционной системе). В нашем листинге такой таблицы нет, и на время работы программы прерывания придется запретить. Запрет всех аппаратных прерываний осуществляется командой *cli* (строка 77).

В строках 78, 79 в порт 70h засылается код 80h, который запрещает немаскируемые прерывания (которые не запрещаются командой *cli*).

В строках 81...83 осуществляется перевод процессора в защищенный режим. Этот перевод можно выполнить различными способами. В рассматриваемом листинге для этого используется команда *mov*. Сначала содержимое управляющего *CR0* регистра считывается в аккумулятор *EAX*, затем его младший бит устанавливается в 1 с помощью команды *or*, затем содержимое аккумулятора опять записывается в управляющий регистр *CR0* с уже модифицированным младшим битом. Все последующие команды выполняются уже в защищенном режиме.

Хотя защищенный режим установлен, однако действия по настройке системы еще не закончены. Действительно, во всех используемых в программе сегментных регистрах хранятся не селекторы дескрипторов сегментов, а базовые сегментные адреса, не имеющие смысла в защищенном режиме.

Отсюда можно сделать вывод, что после перехода в защищенный режим программа не должна работать, так как в регистре *CS* пока еще нет селектора сегмента команд, и процессор не может обращаться к этому сегменту. В действительности это не совсем так.

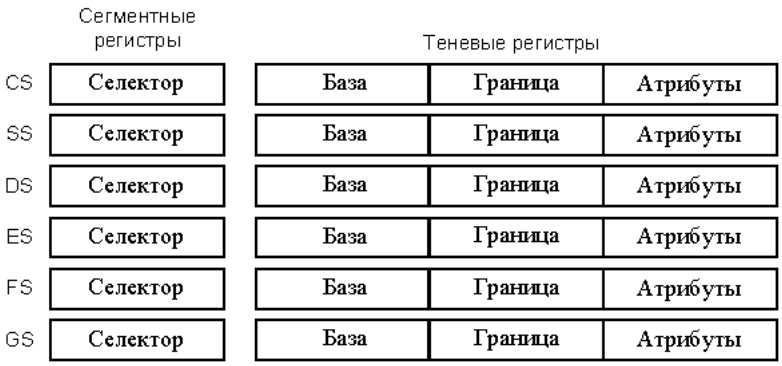


Рисунок 3.11 – Сегментные и теневые регистры

В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора (рисунок 3.11). Теневые регистры недоступны программисту. Они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор загружает соответствующий сегментный регистр. Таким образом, в защищенном режиме программист имеет дело с селекторами, т.е. номерами дескрипторов, а процессор – с самими дескрипторами, хранящимися в теневых регистрах. Именно содержимое теневого регистра (в первую очередь, линейный адрес сегмента) определяет область памяти, к которой обращается процессор при выполнении конкретной команды.

В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным базовым адресом сегмента, полученным путем умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса, и программа будет выполняться правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла.

Тем не менее, после перехода в защищенный режим, прежде всего, следует загрузить в используемые сегментные регистры (и, в частности, в регистр *CS*) селекторы соответствующих сегментов. Это позволит процессору правильно заполнить все поля теневых регистров из таблицы дескрипторов. Пока эта операция не выполнена, некоторые поля теневых регистров (в частности, границы сегментов) могут содержать неверную информацию.

Загрузить селекторы в сегментные регистры *DS*, *SS* и *ES* не представляет труда (строки 93 – 101). Но как загрузить селектор в регистр *CS*? Для этого можно воспользоваться искусственно сконструированной командой дальнего перехода, которая, как известно, приводит к смене содержимого и *IP*, и *CS*. Строки 89 – 91 демонстрируют эту методику. В реальном режиме мы поместили бы во второе слово адреса сегментный адрес сегмента команд, в защищенном же мы записываем в него селектор этого сегмента (число 16).

Команда дальнего перехода, помимо загрузки в *CS* селектора, выполняет еще одну функцию – она очищает очередь команд в блоке предвыборки команд процессора. Как известно, в современных процессорах с целью повышения скорости выполнения программы используется конвейерная обработка команд программы, позволяющая совместить во времени фазы их обработки. Одновременно с выполнением текущей (первой) команды осуществляется выборка операндов следующей (второй), дешифрация третьей и выборка из памяти четвертой команды. Таким образом, в момент перехода в защищенный режим уже могут быть расшифрованы несколько следующих команд и выбраны из памяти их операнды. Однако эти действия выполнялись, очевидно, по правилам реального, а не защищенного режима, что может привести к нарушениям в работе программы. Команда перехода очищает очередь предвыборки, заставляя процессор заполнить ее заново уже в защищенном режиме.

Следующий фрагмент листинга программы (строки 100 – 113) является чисто иллюстративным. В нем инициализируется (по правилам защищенного режима!) сегментный регистр *ES* и в видеобуфер выводится сообщение *'A message in protected mode'* (зеленым цветом на синем фоне), так, что оно располагается в середине пятой строки экрана, чем подтверждается правильное функционирование программы в защищенном режиме.

Как уже отмечалось выше, для того, чтобы не нарушить работоспособность DOS, процессор следует вернуть в реальный режим, после чего можно будет завершить программу обычным образом. Перейти в реальный режим можно разными способами. Можно, например, осуществить

сброс процессора, заслав команду FEh в порт 64h контроллера клавиатуры. Эта команда возбуждает сигнал на одном из выводов контроллера клавиатуры, который, в конечном счете, приводит к появлению сигнала сброса на выводе RESET микропроцессора. Этот способ (единственный для 286 процессора) неудобен тем, что для выполнения сброса необходимо несколько микросекунд. Если выполнять таким образом переход в реальный режим достаточно часто, можно потерять много времени.

В нашей программе переход в реальный режим осуществляется простым сбросом младшего бита в управляющем регистре *CR0*.

Если просто перейти в реальный режим сбросом бита 0 в регистре *CR0*, в теневых регистрах останутся дескрипторы защищенного режима, и при первом же обращении к любому сегменту программы возникнет исключение общей защиты, так как ни один из определенных ранее сегментов не имеет границы FFFh. Так как обработка исключений не производится, произойдет сброс процессора и перезагрузка компьютера, так как в данном случае не настраивался байт состояния перезагрузки, и не заполнялись соответствующие ячейки области данных BIOS. Следовательно, перед переходом в реальный режим необходимо исправить дескрипторы всех используемых сегментов: кодов, данных, стека и видеобуфера. Сегментные регистры *FS* и *GS* не использовались, поэтому о них можно не заботиться.

В строках 118 – 121 в поля границ всех четырех дескрипторов записывается значение FFFFh, а в строках 124 – 129 выполняется загрузка селекторов в сегментные регистры, что приводит к перезаписи содержимого теневых регистров. Так как сегментный регистр *CS* программно недоступен, его загрузку приходится опять выполнять с помощью искусственно сформированной команды дальнего перехода (строки 133 – 135).

После настройки всех использованных в защищенном режиме сегментных регистров, можно сбрасывать бит 0 управляющего регистра *CR0* (строки 137 – 139). После перехода в реальный режим опять надо выполнить искусственно сформированную команды дальнего перехода для того, чтобы очистить очередь команд в блоке предвыборки процессора и загрузить в регистр *CS* вместо хранящегося там селектора обычный сегментный адрес регистра команд (строки 140 – 142).

Искусственно сформированная команда дальнего перехода передает управление на метку *return*.

Теперь процессора опять работает в реальном режиме. При этом, хотя в сегментных регистрах *DS, ES* и *SS* остались недействительные для реального режима селекторы, программа пока работает корректно, так как в теневых регистрах находятся правильные линейные адреса (оставшиеся от

защищенного режима) и законные для реального режима границы (загруженные в строках 118 – 121). Однако, если в программе встретится любая команда сохранения или восстановления какого-либо сегментного регистра, нормальное выполнение программы нарушится, так как в сегментном регистре окажется не сегментный адрес, как это должно быть в реальном режиме, а селектор. Это значение будет трактоваться процессором как сегментный адрес, что приведет в дальнейшем к неверной адресации соответствующего сегмента.

Если даже в оставшемся тексте программы и нет команд чтения или записи сегментных регистров, неприятности все равно возникнут, так как простой вызов DOS'овского прерывания int 21h приведет к этим неприятностям, так как диспетчер DOS выполняет сохранение и восстановление регистров (в том числе и сегментных) при выполнении функций DOS.

Поэтому после перехода в реальный режим необходимо загрузить в используемые далее сегментные регистры соответствующие сегментные адреса. В строках 148 – 151 в регистры *DS* и *SS* записываются сегментные адреса соответствующих сегментов.

При рассмотренном варианте возврата в реальный режим (без сброса процессора) не надо сохранять кадр стека, так как содержимое регистра *SP* в этом случае не разрушается, а регистр *SS* мы уже инициализировали.

Для восстановления работоспособности системы следует также разрешить прерывания (маскируемые – строка 155, немаскируемые – строки 156, 157), после чего программа может продолжаться уже в реальном режиме. В рассмотренном листинге для проверки работоспособности системы в этом режиме на экран выводится сообщение ‘*Real mode now*’ с помощью функции DOS 09h. Для наглядности в сообщение включены Esc последовательности для смены цвета символов и фона (красные символы на зеленом фоне). Осуществлена смена цвета символов и фона может быть лишь в том случае, если в DOS установлен драйвер ANSI.SYS. Если после перехода в реальный режим при установленном драйвере ANSI.SYS сообщение будет выведено без изменения цветов, это может говорить об ошибках защищенного режима.

Перед завершением программы, она ожидает ввода с клавиатуры (функция 0 прерывания *int* 16h), чтобы можно было успеть увидеть содержимое экрана.

Программа завершается обычным образом функцией DOS 4Ch. Нормальное завершение программы и переход в DOS тоже в какой-то мере свидетельствует о ее правильности.

3.3 Вопросы для самопроверки

1. Какие режимы работы поддерживают 32-разрядные процессоры x86?
2. Какие регистры в 32-разрядных микропроцессорах x86 являются 16-разрядными?
3. Какие новые флаги добавились у 32-разрядных микропроцессоров x86?
4. Какие разряды управляющего регистра *CR0* микропроцессора указывают состояние и режимы работы процессора?
5. Что такое бит страничного преобразования?
6. Что такое бит сопроцессора?
7. Для чего нужен бит переключения задачи?
8. Что такое бит эмуляции сопроцессора?
9. Для чего нужен бит присутствия сопроцессора?
10. Что такое бит разрешения защиты?
11. Какие регистры микропроцессора используются для поддержки страничного преобразования?
12. Что такое регистры системных адресов?
13. Для чего нужен регистр таблицы глобальных дескрипторов?
14. Для чего нужен регистр таблицы дескрипторов прерываний?
15. Для чего нужен регистр таблицы локальных дескрипторов?
16. Для чего нужен регистр состояния задачи?
17. Какие действия надо выполнить, чтобы в компилируемой программе можно было использовать 32-разрядные операнды?
18. Какие команды появились в 32-разрядных микропроцессорах (привести примеры)?
19. Каково главное ограничение реального режима работы процессора?
20. Какие дополнительные возможности появляются в защищенном режиме работы микропроцессора?

Тема 4 Использование 32-разрядной адресации в реальном режиме

Большое количество процессоров, используемых в настоящее время, ставит перед программистами проблемы оптимального использования ресурсов конкретного процессора в своих разработках. У изготовителей микропроцессоров стало традицией публиковать описания регистров и команд через Интернет в виде pdf-файлов, но не давать при этом рекомендаций по их применению. Хорошо, если из названия (или описания) можно сделать

совершенно определенные выводы о назначении команды или регистра. А если нет?

Столь же вредная традиция — не описывать в общедоступной документации режимы работы, которых современные процессоры имеют великое множество. Безусловным чемпионом в этой области является Intel — значительная часть потенциальных возможностей процессоров класса Pentium и последующих модификаций не используется потребителями, поскольку эти возможности в документации только упоминаются, но не рассматриваются. Программистам приходится искать наработки энтузиастов, которые тратят свое время на углубленное исследование режимов работы процессоров и применения конкретных, плохо описанных изготовителями, команд и регистров процессора [9].

4.1 Линейная адресация данных в реальном режиме DOS

В литературе по программированию описано три режима работы микропроцессоров серии 80x86 — реальный режим (режим совместимости с архитектурой 8086), защищенный режим и режим виртуальных процессоров 8086 (являющийся неким подвидом защищенного режима).

Основной недостаток реального режима состоит в том, что адресное пространство имеет размер всего в 1 Мбайт и при этом сегментировано — «нарезано» на кусочки размером по 64 Кбайт. Одного мегабайта очень мало для современных ресурсоемких прикладных программ (текстовых и графических редакторов, геоинформационных систем, систем проектирования и т. д.), а сегментация не позволяет нормально работать с видеопамятью и большими массивами данных.

Что можно сказать о защищенном и виртуальном режимах? Многие книги и учебники по микропроцессорам Intel *заканчиваются* главой «Переход в защищенный режим». Недостаток этого режима — необходимость *заново* создавать программное обеспечение для работы с периферийными устройствами на низком уровне, то есть фактически полностью переписывать *все* основные функции DOS. Можно, конечно, использовать Windows, но эта операционная система предназначена для офисных целей и плохо адаптируется к решению задач оперативного управления техническими системами. Кроме того, Windows забирает для собственных нужд изрядную часть ресурсов компьютера и ограничивает доступ к периферийным устройствам.

В некоторых случаях универсальные многозадачные операционные системы типа Windows и Unix неприменимы по причинам, не относящимся напрямую к области вычислительной техники. Первая причина — лицензионные соглашения между изготовителями и потребителями

программ. Прочтите внимательно любую лицензию: разработчик программы не несет ответственности *ни за что*. Следовательно, за все сбои и неисправности расплачивается потребитель. Например, за аварию в системе управления транспортом разработчикам этой системы придется отвечать по статьям Уголовного кодекса. Что касается систем военного назначения, то вообще сомнительно, что на таких лицензионных условиях какая-либо программа может быть официально принята в эксплуатацию на территории России.

Вторая причина — огромный объем универсальных операционных систем — десятки миллионов строк на языках высокого уровня! Полностью протестировать такие системы невозможно — у фирмы Microsoft, например, хватает сил только на доскональную проверку небольшого ядра Windows! Тем более на это не способен потребитель, у которого нет всей документации. Даже в случае открытой системы типа Linux, если документация есть и все исходные коды доступны — попробуйте *доказать* военным или банкирам, что в системе нет скрытых ловушек и «черного хода»!

Создать собственную программу для переключения в защищенный режим и работы в нем — непростая задача. При работе с аппаратурой в защищенном режиме программист должен четко понимать, какими возможностями аппаратуры пользоваться опасно. Например, приводимые в учебниках листинги программ для защищенного режима часто проявляют несовместимость с определенными конфигурациями оборудования, поскольку их авторы не имели достаточно широкой лабораторной базы для тестирования. Дело в том, что периферийные устройства всегда имеют какие-нибудь нестандартные особенности, добавляемые их изготовителями в рекламных целях. При работе в реальном режиме DOS такие особенности не применяются и потому никак не проявляются. Однако они могут показать себя с самой неприятной стороны при переключении в защищенный режим, когда программисту приходится перенастраивать периферийные устройства на новую модель организации оперативной памяти, перезаписывая при этом множество различных регистров аппаратуры. Возможно две ситуации: либо в стандартных регистрах некоторые разряды применяются нестандартным образом, но программисту об этом ничего не известно, либо вообще имеются какие-либо дополнительные регистры, не описанные в документации, но влияющие на режим работы системы. Возникает абсурдная ситуация: простой (реальный) режим работы задается процедурами BIOS фирмы-изготовителя системной платы, которая обычно хорошо осведомлена об особенностях применяемого на этой плате чипсета, а программы для

перехода в защищенный режим вынуждены писать совершенно посторонние люди, не располагающие документацией в полном объеме. В BIOS включено некоторое количество процедур для работы в защищенном режиме, но они охватывают лишь часть необходимых операций.

Вообще говоря, изобилие управляющих регистров в современных персональных компьютерах (их общее количество достигает нескольких тысяч) — явление совершенно ненормальное, теоретически приводящее к увеличению количества возможных режимов работы до бесконечности. Поскольку протестировать функционирование системы в миллиардах различных режимов технически невозможно, разработчики программного обеспечения не могут использовать дополнительные средства, и ограничены несколькими общепринятыми (стандартными) режимами. Чтобы убедиться в этом, достаточно сравнить полный набор команд любого периферийного устройства с реально используемым (например, в BIOS) подмножеством команд данного набора. Большая часть регистров в настоящее время в принципе не нужна — установкой режима работы периферийного устройства должен заниматься его встроенный специализированный процессор, а не центральный процессор компьютера. Однако переход на новые технологии произойдет, вероятно, только после очередного кризиса в развитии компьютерной индустрии, а пока что приходится приспосабливаться к сложившейся ситуации.

Изложенные выше причины приводят к тому, что программисты вынуждены искать различные обходные пути. Один из возможных приемов — использование линейной адресации памяти. Линейная адресация — это наиболее простой, с точки зрения программиста, способ работы непосредственно с аппаратурой ЭВМ (логические адреса при этом совпадают с физическими). Различия в организации памяти в реальном, защищенном и линейном режимах работы процессора иллюстрирует рисунок 4.1.

Линейную адресацию можно использовать в специализированных программах, активно эксплуатирующих ресурсы ЭВМ — как в компьютерных играх, так и в системах автоматики, измерительных системах, системах управления, связи и т. п. Применение линейной адресации целесообразно в том случае, если проектируемая система предназначена для выполнения ограниченного, заранее известного набора функций и требует высокого быстродействия и надежности. Разработчики процессоров начали внедрять линейную адресацию (в качестве одного из возможных режимов работы) при переходе с 16-разрядной архитектуры на 32-разрядную. Фирма Intel ввела такой режим в процессоре 80386, после чего он стал фактически стандартным (поддерживается не только всеми последующими моделями, но и всеми клонами архитектуры x86), однако остался недокументированным (почти не

описан в литературе и не рассматривается в фирменном руководстве по программированию).

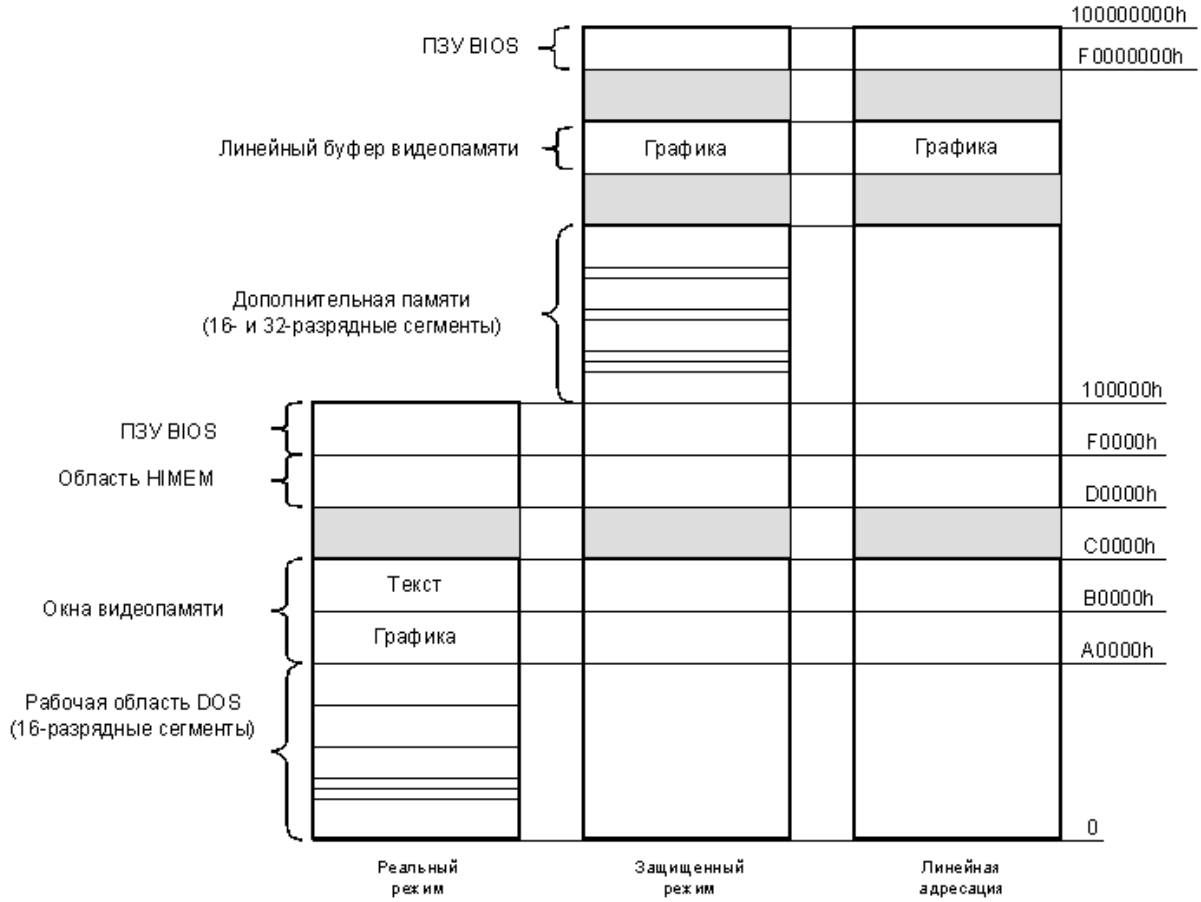


Рисунок 4.1 – Организация адресного пространства в реальном, защищенном и линейном режимах работы процессора x86

Для пользователей обычных персональных компьютеров линейная адресация в чистом виде интереса не представляет по тем же причинам, что и защищенный режим: DOS и BIOS функционируют только в реальном режиме с 64-килобайт-ными сегментами, и при переходе в любой другой режим программист оказывается один на один с аппаратурой ЭВМ — без документации. Однако кроме чистых режимов процессоры Intel способны работать и в режимах гибридных. Еще в 1989 году Томас Роден (Thomas Roden) предложил использовать интересную комбинацию сегментной (для кода и данных) и линейной (только для данных) адресации [2]. Предложенный им метод позволяет, находясь в обычном режиме DOS, работать со всей доступной памятью в пределах четырехгигабайтного адресного пространства процессора Intel 80386. Чтобы включить режим линейной адресации данных, необходимо снять ограничения на размер сегмента в теневом регистре, соответствующем одному из дополнительных сегментных регистров FS или G5 (при необходимости описание архитектуры процессора Pentium можно найти в

документации [3-5], размещенной в Интернете на сервере Intel для разработчиков). Через избранный регистр можно обращаться к любой области памяти с помощью прямой адресации или используя в качестве индексного любой 32-разрядный регистр общего назначения. После снятия ограничения запись в выделенный для линейной адресации сегментный регистр выполнять нельзя, иначе нарушится информация в соответствующем ему теневом регистре (предел сегмента сохранится, но начальный адрес будет перезаписан новым значением). Однако стандартные компиляторы и функции DOS с регистрами FS и GS не работают, и соответственно, при вызове процедур эти регистры можно вообще «не трогать» — их не нужно сохранять и восстанавливать. Достаточно один раз снять ограничение на размер адресного пространства, и после выхода из программы (до перезагрузки компьютера) линейную адресацию можно будет использовать из любой другой программы DOS, как поступил в своем листинге Томас Роден.

Рассмотрим более подробно процедуру переключения одного из дополнительных сегментных регистров в режим линейной адресации. Каждый сегментный регистр, как указано в документации [5], состоит из видимой и невидимой (теневой) частей. Информацию в видимую часть можно записывать напрямую при помощи обычных команд пересылки данных (M0V и др.), а для записи в невидимую часть применяются специальные команды, которые доступны только в защищенном режиме. Теневая часть представляет собой так называемый дескриптор (описатель) сегмента, длина которого равна 8 байтам.

При переходе от 16-разрядной архитектуры к 32-разрядной (то есть от i286 к i386) разработчики нового процессора попытались сохранить совместимость снизу вверх по структуре системных регистров, в результате чего дескрипторы сегментов приобрели довольно уродливый (с точки зрения технической эстетики) вид — поля предела и базового адреса разделены на несколько частей. Кроме того, поле предела оказалось ограничено 20 разрядами, что вынудило разработчиков применить еще один радиолюбительский трюк — ввести бит гранулярности G, чтобы можно было задавать размер сегмента, превышающий 16 Мбайт.

Формат дескриптора сегмента показан на рисунке 4.2. Дескриптор состоит из следующих полей.

- Базовый адрес — 32-разрядное поле, задающее начальный адрес сегмента (в линейном адресном пространстве).
- Предел сегмента — 20-разрядное поле, которое определяет размер сегмента в байтах или 4-килобайтных страницах (в зависимости от значения бита гранулярности G). Поле предела содержит значение,

которое должно быть на единицу меньше реального размера сегмента в байтах или страницах.

- Тип — 4-разрядное поле, определяющее тип сегмента и типы операций, которые допустимо с ним выполнять.
- Бит S — признак системного объекта (0 — дескриптор описывает системный объект, 1 — назначение сегмента описывается полем типа).
- DPL — 2-разрядное поле, определяющее уровень привилегий описываемого дескриптором сегмента.

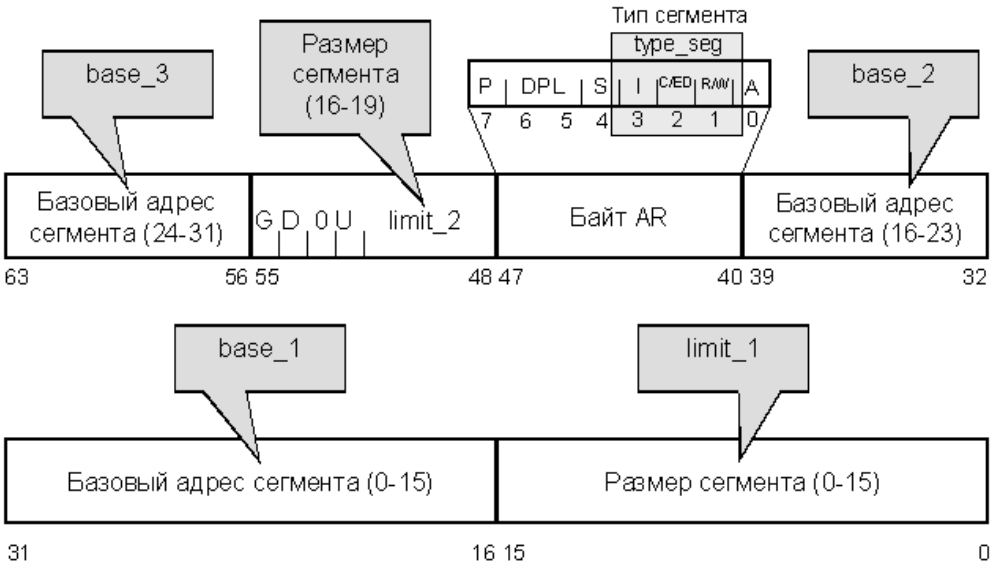


Рисунок 4.2 – Формат дескриптора сегмента

- Бит P — признак присутствия сегмента в оперативной памяти компьютера (0 — сегмент «сброшен» на диск, 1 — сегмент присутствует в оперативной памяти).
- Бит AVL — свободный (available) бит, который может использоваться по усмотрению системного программиста.
- Бит D — признак используемого по умолчанию режима адресации данных (0 — 16-разрядная адресация, 1 — 32-разрядная).
- Бит G — гранулярности сегмента (0 — поле предела задает размер сегмента в байтах, 1 — в 4-килобайтных страницах).

В нашем случае признак используемого по умолчанию режима адресации данных D можно установить в 0 (использовать по умолчанию 16-разрядные операнды), но особой роли его значение не играет — в смешанном режиме сегментно-линейной адресации при работе с линейным сегментом строковые команды, использующие значение этого разряда, применять нельзя. Бит гранулярности G должен быть установлен в 1, чтобы обеспечить охват всего адресного пространства процессора.

Рисунок 4.3 – Формат прав доступа для сегмента данных

Для сегментов данных формат байта прав доступа (включающего поле типа сегмента) имеет вид, показанный на рисунке 8.3. Как видно из рисунка, поле S для сегментов данных должно быть установлено в 1, а старший разряд поля типа должен иметь значение 0. Поля P и DPL уже упоминались выше. Бит присутствия сегмента P следует установить в 1 (сегмент присутствует в памяти), а в поле DPL нужно установить максимальный уровень привилегий (значение 00). Бит расширения вниз ED для сегментов данных имеет значение 0 (в отличие от стековых сегментов, для которых ED=1). Бит разрешения записи W следует установить в единицу, чтобы можно было не только считывать, но и записывать информацию в сегмент. Бит A фиксирует обращение к сегменту и автоматически устанавливается в единицу всякий раз, когда процессор производит операции считывания или записи с сегментом, описываемым данным дескриптором. При инициализации регистра бит A можно сбросить в 0.

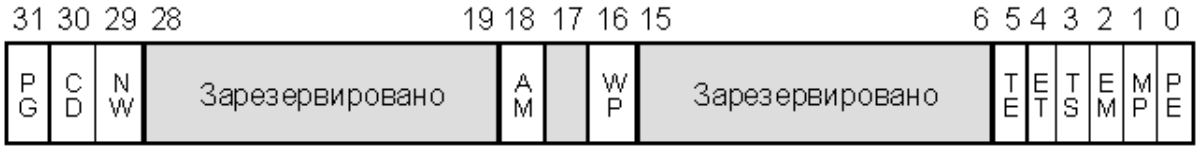


Рисунок 4.4 – Формат управляющего регистра CR0

Осуществить загрузку теневых регистров можно только в защищенном режиме. Для переключения режимов работы процессора используется регистр управления CR0, формат которого показан на рисунке 4.4 [15]. Регистр CR0 содержит флаги, отражающие состояние процессора и управляющие режимами его работы. Назначение флагов следующее.

- *PE* (Protection Enable) —разрешение защиты Установка этого флага инструкцией *LMSW* или *LOAD CR0* переводит процессор в защищенный режим. Сброс флага (возврат в реальный режим) возможет только по инструкции *LOAD CR0.* Сброс бита *PE* является частью довольно длинной последовательности инструкций, подготавливающих корректное переключение в реальный режим.
- *MP* (Monitor Processor Extension) — мониторинг математического сопроцессора. Позволяет вызывать исключение *#NM* по каждой

команде *WAIT* при *TS=1*. При выполнении программ для процессоров 286/287 и 386/387 на процессорах 486 DX и старше бит *MP* должен быть установлен.

- *EM* (Processor Extension Emulated) — эмуляция математического сопроцессора. Установка этого флага вызывает исключение *#NM* при каждой команде, относящейся к сопроцессору, что позволяет прозрачно осуществлять его программную эмуляцию.
- *TS* (Task Switched) — признак переключения задачи (флаг устанавливается в 1 при каждом переключении задач и проверяется перед выполнением команд математического сопроцессора).
- *ET* (Extension Type) — индикатор поддержки набора инструкций математического сопроцессора (0 — выключена, 1 — включена). В процессорах P6 флаг всегда установлен в 1.
- *NE* (Numeric Error) — встроенный механизм контроля ошибок математического сопроцессора (0 — выключен, 1 —включен).
- *WP* (Write Protect) — защита от записи информации в страницы уровня пользователя из процедур супервизора (0 — выключена, 1 — включена).
- *AH* (Alignment Mask) — разрешение контроля выравнивания (контроль выравнивания выполняется только на уровне привилегий 3 при *AM*=1 и флаге *AC*=1. (0 — запрещен, 1 — разрешен).
- *NW* (Not Writethrough) — запрещение сквозной записи и циклов аннулирования. (0 — сквозная запись разрешена, 1 — запрещена).
- *CD* (Cache Disable) — запрет заполнения кэш-памяти (попадание в ранее заполненные строки при этом обслуживаются кэшем). (0 — использование кэш-памяти разрешено, 1 — запрещено).
- *PG* (Paging) — включение страничного преобразования памяти (0 — запрещено, 1 — разрешено).

Набор подпрограмм, необходимых для переключения сегментного регистра GS в режим линейной адресации, показан в листинге 8.1 [9]. Как сказано выше, перезапись содержимого теневого регистра процессора возможна только в защищенном режиме, а переход в этот режим, как видно из листинга, требует ряда дополнительных операций, выполняемых процедурой *Initialization*. В частности, нужно перенастроить на специально выделенные в кодовом сегменте области данных регистры DS, SS и SP. В момент перенастройки регистров стека должны быть запрещены прерывания, поскольку некоторые обработчики прерываний пишут информацию в стек прерываемой программы.

Процедура **SetLAddrModeForGS**, непосредственно осуществляющая перенастройку регистра GS в режим линейной адресации, воспроизводит (с незначительными изменениями) метод Родена. Прежде чем осуществить переключение, нужно вначале подготовить таблицу *GDT* (настроить на текущие сегменты кода и данных) и загрузить ее. Затем нужно войти в защищенный режим — установить в единицу бит *PE* регистра CR0, а остальные разряды сохранить без изменений (в том виде, в котором они находились при работе в реальном режиме). В защищенном режиме необходимо перезагрузить сегментные регистры, сняв при этом ограничения с GS, и сразу же вернуться в реальный режим DOS, сбросив в ноль бит *PE*. Длительное пребывание в защищенном режиме нежелательно, поскольку переключение в него выполнялось по упрощенной схеме: таблица прерываний не создавалась, а сами прерывания были просто отключены.

После выполнения процедуры **SetLAddrModeForGS** обязательно следует отменить замыкание адресного пространства, то есть разблокировать адресную линию *A20*, которая управляется контроллером клавиатуры. Для этого необходимо послать в порт *A* контроллера соответствующую команду. Посылка команды осуществляется при помощи **Enable_A20** и **Wait8042Buffer Empty**.

Подпрограмма, устанавливающая режим линейной адресации данных

```
; Порт, управляющий запретом немаскируемых прерываний
CMOS_ADDR       equ  0070h
CMOS_DATA       equ  0071h
; Селекторы сегментов
SYS_PROT_CS     equ  0008h
SYS_REAL_SEG    equ  0010h
SYS_MONDO_SEG   equ  0018h

CODESEG
;*********************************************
;* ВКЛЮЧЕНИЕ РЕЖИМА ЛИНЕЙНОЙ АДРЕСАЦИИ ПАМЯТИ *
;*       (процедура параметров не имеет)     *
;*********************************************
PROC Initialization NEAR
        pushad
; Сохранить значения сегментных регистров в
; реальном режиме (кроме GS)
        mov     [CS:Save_SP],SP
        mov     AX,SS
        mov     [CS:Save_SS],AX
        mov     AX,DS
```

```
        mov     [CS:Save_DS],AX
; (работаем теперь только с кодовым сегментом)
        mov     AX,CS
        mov     [word ptr CS:Self_Mod_CS],AX
        mov     DS,AX
        cli
        mov     SS,AX
        mov     SP,offset Local_Stk_Top
        sti

; Установить режим линейной адресации
        call    SetLAddrModeForGS

; Восстановить значения сегментных регистров
        cli
        mov     SP,[CS:Save_SP]
        mov     AX,[CS:Save_SS]
        mov     SS,AX
        mov     AX,[CS:Save_DS]
        mov     DS,AX
        sti

; Разрешить работу линии A20
        call    Enable_A20
        popad
        ret
ENDP Initialization


; Область сохранения значений сегментных регистров
Save_SP DW ?
Save_SS DW ?
Save_DS DW ?
; Указатель на GDT
GDTPtr  DQ ?
; Таблица дескрипторов сегментов для
; входа в защищенный режим
GDT DW 00000h,00000h,00000h,00000h ;не используется
    DW 0FFFFh,00000h,09A00h,00000h ;сегмент кода CS
    DW 0FFFFh,00000h,09200h,00000h ;сегмент данных DS
    DW 0FFFFh,00000h,09200h,0008Fh ;сегмент GS
; Локальный стек для защищенного режима
; (организован внутри кодового сегмента)
label GDTEnd word
        DB 255 DUP(0FFh)
Local_Stk_Top DB (0FFh)
```

```
;*********************************************
;*        ОТМЕНИТЬ ПРЕДЕЛ СЕГМЕНТА GS        *
;* Процедура изменяет содержимое теневого    *
;* регистра GS таким образом, что становится *
;* возможной линейная адресация через него   *
;* 4 Gb памяти в реальном режиме             *
;*********************************************
PROC SetLAddrModeForGS near
; Вычислить линейный адрес кодового сегмента
        mov     AX,CS
        movzx   EAX,AX
        shl     EAX,4   ;умножить номер параграфа на 16
        mov     EBX,EAX ;сохранить линейный адрес в EBX
; Занести младшее слово линейного адреса в дескрипторы
; сегментов кода и данных
        mov     [word ptr CS:GDT+10],AX
        mov     [word ptr CS:GDT+18],AX
        ; Переставить местами старшее и младшее слова
        ror     EAX,16
; Занести биты 16-23 линейного адреса в дескрипторы
; сегментов кода и данных
        mov     [byte ptr CS:GDT+12],AL
        mov     [byte ptr CS:GDT+20],AL
; Установить предел (Limit) и базу (Base) для GDTR
        lea    ax,[GDT] ;*************
        movzx  eax,ax    ;*************
        add     EBX,EAX;   offset GDT
        mov [word ptr CS:GDTPtr],(offset GDTEnd-GDT-1)
        mov     [dword ptr CS:GDTPtr+2],EBX
; Сохранить регистр флагов
        pushf
; Запретить прерывания, так как таблица прерываний IDT
; не сформирована для защищенного режима
        cli
; Запретить немаскируемые прерывания NMI
        in      AL,CMOS_ADDR
        mov     AH,AL
        or      AL,080h      ;установить старший разряд
        out     CMOS_ADDR,AL ;не затрагивая остальные
        and     AH,080h
        ; Запомнить старое состояние маски NMI
        mov     CH,AH
; Перейти в защищенный режим
        lgdt    [fword ptr CS:GDTPtr]
```

```
        mov     BX,CS    ;запомнить сегмент кода
        mov     EAX,CR0
        or      AL,01b   ;установить бит PE
        mov     CR0,EAX  ;защита разрешена
        ; Безусловный дальний переход на метку SetPMode
        ; (очистить очередь команд и перезагрузить CS)
                DB      0EAh
                DW      (offset SetPMode)
                DW      SYS_PROT_CS
SetPMode:
        ; Подготовить границы сегментов
        mov     AX,SYS_REAL_SEG
        mov     SS,AX
        mov     DS,AX
        mov     ES,AX
        mov     FS,AX
        ; Снять ограничения с сегмента GS
        mov     AX,SYS_MONDO_SEG
        mov     GS,AX
; Вернуться в реальный режим
        mov     EAX,CR0
        and     AL,11111110b ;сбросить бит PE
        mov     CR0,EAX      ;защита отключена

        ; Безусловный дальний переход на метку SetRMode
        ; (очистить очередь команд и перезагрузить CS)
            DB 0EAh
            DW (offset SetRMode)
Self_Mod_CS DW ?

SetRMode:
        ; Регистры стека и данных
        ; настроить на сегмент кода
        mov     SS,BX
        mov     DS,BX
        ; Обнулить дополнительные сегментные
        ; регистры данных (GS не трогать!)
        xor     AX,AX
        mov     ES,AX
        mov     FS,AX
        ; Возврат в реальный режим,
        ; прерывания снова разрешены
        in      AL,CMOS_ADDR
        and     AL,07Fh
        or      AL,CH
        out     CMOS_ADDR,AL
```

```
        popf
        ret
ENDP SetLAddrModeForGS


;****************************************
;* Разрешить работу с памятью выше 1 Мб *
;****************************************
PROC Enable_A20 near
        call    Wait8042BufferEmpty
        mov     AL,0D1h ;команда управления линией A20
        out     64h,AL
        call    Wait8042BufferEmpty
        mov     AL,0DFh ;разрешить работу линии
        out     60h,AL
        call    Wait8042BufferEmpty
        ret
ENDP Enable_A20


;************************************************
;*   ОЖИДАНИЕ ОЧИСТКИ ВХОДНОГО БУФЕРА I8042     *
;* При выходе из процедуры:                     *
;* флаг ZF установлен - нормальное завершение,  *
;* флаг ZF сброшен - ошибка тайм-аута.          *
;************************************************
proc Wait8042BufferEmpty near
        push    CX
        mov     CX,0FFFFh  ;задать число циклов
@@kb:   in      AL,64h     ;получить статус
        test    AL,10b     ;буфер i8042 свободен?
        loopnz  @@kb       ;если нет, то цикл
        pop     CX
        ; (если при выходе сброшен флаг ZF - ошибка)
        ret
endp Wait8042BufferEmpty
ENDS
```

ВНИМАНИЕ! Как уже было сказано, после выхода из защищенного режима нельзя перезаписывать регистр GS, иначе будет полностью или частично стерта информация в соответствующем теневом регистре. В частности, нельзя выполнять операции сохранения/восстановления содержимого регистра при помощи команд работы со стеком *push* и *pop*.

При использовании нестандартных режимов работы возникают определенные трудности в процессе отладки программ: стандартные

программы-отладчики становятся неудобными. Во многих случаях, однако, достаточно использовать простую отладочную печать. В листинге 8.2 [9] приведена подпрограмма ShowRegs, отображающая на экране содержимое регистров общего назначения, сегментных регистров, регистра флагов и регистра CR0. Недостаток этого упрощенного подхода заключается в том, что ShowRegs *не сохраняет содержимое видеопамяти.* Однако при использовании линейной адресации программу не трудно усовершенствовать, если есть достаточный запас оперативной памяти: в текстовом режиме для сохранения одной страницы нужно менее 4 Кбайт, а в графическом режиме TrueColor32 с разрешением 1920x1280 требуется уже 9,5 Мбайт.

Отладочная подпрограмма, предназначенная для отображения на экран содержимого регистров процессора

```
DATASEG
label REGROW_386 byte
   DB 0,0,'EAX =',0
   DB 1,0,'EBX =',0
   DB 2,0,'ECX =',0
   DB 3,0,'EDX =',0
   DB 4,0,'ESI =',0
   DB 5,0,'EDI =',0
   DB 6,0,'EBP =',0
   DB 7,0,'ESP =',0
   DB 8,0,'IP  =',0
   DB 9,0,'CS  =',0
   DB 10,0,'DS  =',0
   DB 11,0,'ES  =',0
   DB 12,0,'FS  =',0
   DB 13,0,'GS  =',0
   DB 14,0,'SS  =',0
   DB 16,8,'              AVR  NIOODIT SZ A P C',0
   DB 17,8,'              CMF  TPLFFFF FF F F F',0
   DB 18,0,'Флаги:',0
   DB 20,8,'PCN           A V            NETEMP',0
   DB 21,8,'GDW           M P            ETSMPE',0
   DB 22,0,'CR0:',0
   DB 24,15
   DB 'Для продолжения работы нажмите любую клавишу',0


CODESEG
;***********************************************
;* ВЫВЕСТИ НА ЭКРАН ДАМП РЕГИСТРОВ ПРОЦЕССОРА *
;*       (процедура параметров не имеет)       *
;***********************************************
```

```
PROC ShowRegs FAR
        pushad
        pushfd
        push    DS
        mov     BP,SP
        mov     AX,DGROUP
        mov     DS,AX
; Сохраняем глобальные переменные
        mov     AL,[TextColorAndBackground]
        push    AX
        push    [ScreenString]
        push    [ScreenColumn]
; Очищаем экран
        call    ClearScreen
; Вывести 21 строку текста
        mov     [TextColorAndBackground],YELLOW
        mov     SI, offset REGROW_386
        mov     CX,22
@@GLB:  call    ShowString
        loop    @@GLB

        mov     [TextColorAndBackground],WHITE

        mov     EAX,[BP+34] ;Показать EAX
        mov     [ScreenString],0
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+22] ;Показать EBX
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+30] ;Показать ECX
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+26] ;Показать EDX
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+10] ;Показать ESI
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+6]  ;Показать EDI
        inc     [ScreenString]
        mov     [ScreenColumn],6
```

```
        call    ShowHexDWord
        mov     EAX,[BP+14] ;Показать EBP
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     EAX,[BP+18] ;Показать ESP
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexDWord
        mov     AX,[BP+38]  ;Показать IP
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,[BP+40]  ;Показать CS
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,[BP]     ;Показать DS
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,ES       ;Показать ES
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,FS       ;Показать FS
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,GS       ;Показать GS
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        mov     AX,SS       ;Показать SS
        inc     [ScreenString]
        mov     [ScreenColumn],6
        call    ShowHexWord
        add     [ScreenString],4
        mov     [ScreenColumn],8
        mov     EAX,[BP+2]
        call    ShowBinDWord
        add     [ScreenString],4
        mov     [ScreenColumn],8
        mov     EAX,CR0
        call    ShowBinDWord
; Ожидаем нажатия любого символа на клавиатуре
```

```
        call    GetChar
; Очищаем экран
        call    ClearScreen

; Восстановить глобальные переменные
        pop     [ScreenColumn]
        pop     [ScreenString]
        pop     AX
        mov     [TextColorAndBackground],AL
        pop     DS
        popfd
        popad
        ret
ENDP ShowRegs
ENDS
```

В программе LAddrTest, показанной ниже [9], используются процедуры из листингов 8.1 и 8.2 для включения режима линейной адресации и демонстрации изменения содержимого сегментных регистров, которое при этом происходит (процедура установки линейного режима перезаписывает теневой регистр у регистра GS, а регистры ES и FS просто обнуляет). После выполнения программы режим линейной адресации данных *сохраняется,* и любая другая программа, в том числе написанная на языке высокого уровня, может через GS обращаться к любой области памяти по физическому адресу.

Включение режима линейной адресации

```
IDEAL
P386
LOCALS
MODEL MEDIUM

; Подключить файл мнемонических обозначений
; кодов управляющих клавиш
include "lst_2_03.inc"
; Подключить файл мнемонических обозначений цветов
include "lst_2_05.inc"

SEGMENT sseg para stack 'STACK'
DB 400h DUP(?)
ENDS

DATASEG
; Текстовые сообщения
Text1 DB 0,19,"Включение режима "
      DB "линейной адресации данных",0
```

```
        DB 11,0,"Для просмотра "
        DB "содержимого регистров процессора",0
        DB 12,0,"перед запуском процедуры "
        DB "перехода в режим",0
        DB 13,0,"линейной адресации нажмите "
        DB "любую клавишу.",0
Text2   DB 11,0,"Произведено переключение в "
        DB "режим линейной адресации.",0
        DB 12,0,"Для просмотра содержимого "
        DB "регистров процессора",0
        DB 13,0,"нажмите любую клавишу.",0
Text3   DB 11,0,"После завершения данной "
        DB "программы регистр GS",0
        DB 12,0,"может использовать для "
        DB "линейной адресации",0
        DB 13,0,"любая другая программа.",0
        DB 24,18,"Для выхода из программы "
        DB "нажмите любую клавишу.",0
ENDS


CODESEG
;*****************************
;* Основной модуль программы *
;*****************************
PROC LAddrTest
        mov     AX,DGROUP
        mov     DS,AX
; Установить текстовый режим и очистить экран
        mov     AX,3
        int     10h
; Скрыть курсор - убрать за нижнюю границу экрана
        mov     [ScreenString],25
        mov     [ScreenColumn],0
        call    SetCursorPosition
; Вывести первое текстовое сообщение
; на экран зеленым цветом
        mov     [TextColorAndBackground],LIGHTGREEN
        mov     CX,4
        mov     SI,offset Text1
@@NextString1:
        call    ShowString
        loop    @@NextString1
        ; Ожидать нажатия любой клавиши
        call    GetChar
; Занести контрольное число в дополнительные
```

```
; сегментные регистры данных
        mov     AX,0ABCDh
        mov     ES,AX
        mov     FS,AX
        mov     GS,AX
; Показать содержимое регистров процессора
        call    far ShowRegs

; Установить режим прямой адресации памяти
     call    Initialization

; Вывести второе текстовое сообщение
; на экран голубым цветом
        mov     [TextColorAndBackground],LIGHTCYAN
        mov     CX,3
        mov     SI,offset Text2
@@NextString2:
        call    ShowString
        loop    @@NextString2
        ; Ожидать нажатия любой клавиши
        call    GetChar
; Показать содержимое регистров процессора
        call    far ShowRegs

; Вывести третье текстовое сообщение
; на экран желтым цветом
        mov     [TextColorAndBackground],YELLOW
        mov     CX,4
        mov     SI,offset Text3
@@NextString3:
        call    ShowString
        loop    @@NextString3
        ; Ожидать нажатия любой клавиши
        call    GetChar

; Установить текстовый режим
        mov     ax,3
        int     10h
; Выход в DOS
        mov     AH,4Ch
        int     21h
ENDP LAddrTest
ENDS

; Подключить набор процедур вывода/вывода данных
include "lst_2_02.inc"
```

```
; Подключить подпрограмму, переводящую сегментный
; регистр GS в режим линейной адресации
include "lst_3_01.inc"
; Подключить подпрограмму, отображающую на экране
; содержимое регистров процессора
include "lst_3_02.inc"

END
```

Показанный листинг [9] демонстрирует использование линейной адресации для отображения содержимого памяти компьютера на экране, то есть выдачи дампа памяти. Программа *MemoryDump* позволяет просматривать все адресное пространство, а не только оперативную память. Можно, например, считывать память видеоконтроллера или вообще неиспользуемые области.

Кроме процедур ввода/вывода общего назначения, в *MemoryDump* используются также следующие подпрограммы:

- процедура *ShowASCIIChar* осуществляет вывод символа в ASCII-коде в заданную позицию экрана;
- процедура *HexToBin32* осуществляет перевод числа (введенного с клавиатуры адреса) из шестнадцатеричного кода в двоичный;
- процедура *GetAddressOrCommand* принимает команды, вводимые с клавиатуры (введенное число воспринимается как линейный адрес памяти в шестнадцатеричном коде, нажатие на управляющие клавиши — как команда).

Использование линейной адресации для отображения на экран содержимого оперативной памяти

```
IDEAL
P386
LOCALS
MODEL MEDIUM

; Подключить файл мнемонических обозначений
; кодов управляющих клавиш
include "lst_2_03.inc"
; Подключить файл мнемонических обозначений цветов
include "lst_2_05.inc"

SEGMENT sseg para stack 'STACK'
DB 400h DUP(?)
ENDS

DATASEG
; Текстовые сообщения
```

```
Txt1 DB LIGHTMAGENTA,0,28,"Дамп оперативной памяти",0
     DB YELLOW,2,0,"Адрес:",0
     DB LIGHTGREEN,2,11
     DB "Шестнадцатеричное представление:",0
     DB LIGHTCYAN,2,61,"ASCII-коды:",0
     DB LIGHTRED,21,0,"Введите число "
     DB "или нажмите управляющую клавишу:",0
Txt2 DB 23,0, "Стрелка вниз - следующие 256 байт;",0
     DB 23,35, "Стрелка вверх - предыдущие 256 байт;",0
     DB 24,0, "Enter - завершение ввода адреса;",0
     DB 24,33, "Esc - отмена ввода адреса;",0
     DB 24,60, "F10 - выход.",0
; Количество введенных символов числа
CharacterCounter DB 0
; Позиция для ввода адреса на экране
OutAddress DB 21,47
; Строка для ввода адреса
AddressString DB 9 DUP(0)
; Строка пробелов для "затирания" числа
SpaceString DB 21,47,9 DUP(' '),0
; Начальный адрес
StartAddress DD 0
; Код команды
CommandByte DB 0
ENDS


CODESEG
;*****************************
;* Основной модуль программы *
;*****************************
PROC MemoryDump
        mov     AX,DGROUP
        mov     DS,AX
; Устанавливаем режим прямой адресации памяти
        call    Initialization
; Установить текстовый режим и очистить экран
        mov     AX,3
        int     10h
; Скрыть курсор - убрать за нижнюю границу экрана
        mov     [ScreenString],25
        mov     [ScreenColumn],0
        call    SetCursorPosition
; Вывести текстовые сообщения на экран
        mov     CX,5
        mov     SI,offset Txt1
```

```
@@NextString1:
        call    ShowColorString
        loop    @@NextString1
        mov     [TextColorAndBackground],WHITE
        mov     CX,5
        mov     SI,offset Txt2
@@NextString2:
        call    ShowString
        loop    @@NextString2

; Установить белый цвет символов и черный фон
        mov     [TextColorAndBackground],WHITE
; Отобразить символы-разделители колонок
        mov     AL,0B3h
        mov     [ScreenString],2
        mov     [ScreenColumn],9
        call    ShowASCIIChar
        mov     [ScreenColumn],59
        call    ShowASCIIChar
        mov     [ScreenString],3
        mov     [ScreenColumn],9
        call    ShowASCIIChar
        mov     [ScreenColumn],59
        call    ShowASCIIChar

; Инициализируем переменные
        mov     [StartAddress],0
        mov     [CommandByte],0
; ВНЕШНИЙ ЦИКЛ
@@q0:   mov     EBX,[StartAddress]
        mov     [ScreenString],4
        mov     DX,16
@@q1:   mov     [ScreenColumn],0
; Отобразить линейный адрес первого байта в группе
        mov     [TextColorAndBackground],YELLOW
        mov     EAX,EBX
        call    ShowHexDWord
; Отобразить символ-разделитель колонок
        mov     [TextColorAndBackground],WHITE
        inc     [ScreenColumn]
        mov     AL,0B3h
        call    ShowASCIIChar
        inc     [ScreenColumn]
; Отобразить очередную группу байт
; в шестнадцатеричном коде
        mov     CX,16
```

```
        mov     [TextColorAndBackground],LIGHTGREEN
@@q2:   mov     AL,[GS:EBX]
        call    ShowByteHexCode
        inc     [ScreenColumn]
        inc     EBX
        loop    @@q2
; Отобразить символ-разделитель колонок
        mov     [TextColorAndBackground],WHITE
        mov     AL,0B3h
        call    ShowASCIIChar
        inc     [ScreenColumn]
        ; Вернуться назад на 16 символов
        sub     EBX,16
; Отобразить очередную группу байт в кодах ASCII
        mov     CX,16
        mov     [TextColorAndBackground],LIGHTCYAN
@@q3:   mov     AL,[GS:EBX]
        call    ShowASCIIChar
        inc     EBX
        loop    @@q3
        inc     [ScreenString]
        dec     DX
        jnz     @@q1

        ; Ожидать нажатия любой клавиши
        call    GetAddressOrCommand
        cmp     [CommandByte],F10
        jne     @@q0

@@End:  ; Установить текстовый режим
     mov     ax,3
     int     10h
; Выход в DOS
     mov     AH,4Ch
     int     21h
ENDP MemoryDump


;*****************************************************
;*         ВЫВОД БАЙТА НА ЭКРАН В КОДЕ ASCII         *
;* Подпрограмма выводит содержимое регистра AL в коде *
;* ASCII в указанную позицию экрана.                 *
;* Координаты позиции передаются через глобальные    *
;* переменные ScreenString и ScreenColumn. После     *
;* выполнения операции вывода происходит автомати-   *
;* ческое приращение значений этих переменных.       *
```

```
;*******************************************************
PROC ShowASCIIChar near
        pusha
        push    DS
        push    ES
        mov     DI,DGROUP
        mov     DS,DI
        cld
; Настроить пару ES:DI для прямого вывода в видеопамять
        push    AX
        ; Загрузить адрес сегмента видеоданных в ES
        mov     AX,0B800h
        mov     ES,AX
        ; Умножить номер строки на длину строки в байтах
        mov     AX,[ScreenString]
        mov     DX,160
        mul     DX
        ; Прибавить номер колонки (дважды)
        add     AX,[ScreenColumn]
        add     AX,[ScreenColumn]
        ; Переписать результат в индексный регистр
        mov     DI,AX
        pop     AX
        mov     AH,[TextColorAndBackground]
        stosw

; Подготовка для вывода следующих байтов
        ; Перевести текущую позицию на 2 символа влево
        inc     [ScreenColumn]
        ; Проверить пересечение правой границы экрана
        cmp     [ScreenColumn],80
        jb      @@End
        ; Если достигнута правая граница экрана -
        ; перейти на следующую строку
        sub     [ScreenColumn],80
        inc     [ScreenString]
@@End:  pop     ES
        pop     DS
        popa
        ret
ENDP ShowASCIIChar


;*******************************************************
;* ПЕРЕВОД ЧИСЛА ИЗ ШЕСТНАДЦАТЕРИЧНОГО КОДА В ДВОИЧНЫЙ *
;* DS:SI - число в коде ASCII.                         *
```

```
;* Результат возвращается в EAX.                          *
;*********************************************************
PROC HexToBin32 near
        push    EBX
        push    CX
        push    SI
        cld
        xor     EBX,EBX ;обнуляем накопитель
        xor     CX,CX   ;обнуляем счетчик цифр
@@h0:   lodsb
        ; Проверка на ноль (признак конца строки)
        and     AL,AL
        jz      @@h4
        ; Проверка на диапазон '0'-'9'
        cmp     AL,'0'
        jb      @@Error
        cmp     AL,'9'
        ja      @@h1
        sub     AL,'0'
        jmp short @@h3
@@h1:   ; Проверка на диапазон 'A'-'F'
        cmp     AL,'A'
        jb      @@Error
        cmp     AL,'F'
        ja      @@h2
        sub     AL,'A'-10
        jmp short @@h3
@@h2:   ; Проверка на диапазон 'a'-'f'
        cmp     AL,'a'
        jb      @@Error
        cmp     AL,'f'
        ja      @@Error
        sub     AL,'a'-10
@@h3:   ; Дописать к результату
        ; очередные 4 разряда справа
        shl     EBX,4
        or      BL,AL
        inc     CX
        cmp     CX,8
        jbe     @@h0
        ; Если в числе больше 8 цифр - ошибка
        jmp short @@Error
@@h4:   ; Успешное завершение - результат в EAX
        mov     EAX,EBX
        jmp short @@End
@@Error:; Ошибка - обнулить результат
```

```
        xor     EAX,EAX
@@End:  pop     SI
        pop     CX
        pop     EBX
        ret
ENDP HexToBin32


;************************************************
;* ПРИНЯТЬ С КЛАВИАТУРЫ НОВЫЙ АДРЕС ИЛИ КОМАНДУ *
;************************************************
PROC GetAddressOrCommand near
        pushad
        ; Использовать при выводе белый цвет, черный фон
        mov     [TextColorAndBackground],WHITE
        ; Установить номер строки поля ввода
        mov     [ScreenString],21
@@GetAddressOrCommand:
; Инициализировать переменные
        ; Обнулить счетчик цифр
        mov     [CharacterCounter],0
        ; Очистить строку
        mov     DI,offset AddressString
        mov     [byte ptr DS:DI],0
        ; Очистить позицию ввода (забить пробелами)
        mov     SI,offset SpaceString
        call    ShowString
        ; Установить курсор в позицию ввода
        mov     [ScreenColumn],47
        mov     AL,[CharacterCounter]
        add     [byte ptr ScreenColumn],AL
        call    SetCursorPosition
        ; Ввести цифру или команду
        call    GetChar
        ; Адрес или команда?
        cmp     AL,0
        jz      @@Command
        ; Введена первая цифра числа

; ВВОД АДРЕСА В ШЕСТНАДЦАТЕРИЧНОМ КОДЕ
@@Address:
        ; Проверка на диапазон '0'-'9'
        cmp     AL,'0'
        jb      @@AddressError
        cmp     AL,'9'
        jbe     @@WriteChar
```

```
        ; Проверка на диапазон 'A'-'F'
        cmp     AL,'A'
        jb      @@AddressError
        cmp     AL,'F'
        jbe     @@WriteChar
        ; Проверка на диапазон 'a'-'f'
        cmp     AL,'a'
        jb      @@AddressError
        cmp     AL,'f'
        ja      @@AddressError
@@WriteChar:
        ; Проверяем количество цифр
        cmp     [CharacterCounter],8
        jae     @@AddressError
        inc     [CharacterCounter]
        ; Записываем цифру в число
        mov     [DS:DI],AL
        inc     DI
        ; Передвинуть признак конца строки
        ; в следующий разряд
        mov     [byte ptr DS:DI],0
        ; Отобразить число на экране
        mov     SI,offset SpaceString
        call    ShowString
        mov     SI,offset OutAddress
        call    ShowString
@@GetNextChar:
        ; Отобразить курсор в новой позиции ввода
        mov     [ScreenColumn],47
        mov     AL,[CharacterCounter]
        add     [byte ptr ScreenColumn],AL
        call    SetCursorPosition
        ; Ожидать ввода следующего символа
        call    GetChar
        cmp     AL,0
        jne     @@Address

; Проанализировать код нажатой клавиши
        cmp     AH,B_Esc        ;отмена ввода адреса
        je      @@GetAddressOrCommand

@@TestF10:
        cmp     AH,F10          ;"Выход"
        jne     @@TestRubout
        mov     [CommandByte],AH
        jmp     @@End
```

```
@@TestRubout:
        cmp     AH,B_RUBOUT     ;"Забой"
        jne     @@TestEnter
        cmp     [CharacterCounter],0
        je      @@AddressError
        ; Передвинуть признак конца строки
        ; на разряд влево
        dec     DI
        dec     [CharacterCounter]
        mov     [byte ptr DS:DI],0
        ; Отобразить число на экране
        mov     SI,offset SpaceString
        call    ShowString
        mov     SI,offset OutAddress
        call    ShowString
        jmp     @@GetNextChar

@@TestEnter:
        cmp     AH,B_Enter      ;завершение ввода числа
        jne     @@AddressError
        mov     [CommandByte],AH
        mov     SI,offset AddressString
        call    HexToBin32
        mov     [StartAddress],EAX
        jmp short @@End

@@AddressError:
        call    Beep
        jmp     @@GetNextChar

; ОБРАБОТКА "КОМАНД"
@@Command:
        cmp     AH,F10          ;"Выход"
        jne     @@TestDn
        mov     [CommandByte],AH
        jmp short @@End

@@TestDn:
        cmp     AH,B_DN         ;"Стрелка вниз"
        jne     @@TestUp
        mov     [CommandByte],AH
        add     [StartAddress],256
        jmp short @@End

@@TestUp:
```

```
        cmp     AH,B_UP         ;"Стрелка вверх"
        jne     @@CommandError
        mov     [CommandByte],AH
        sub     [StartAddress],256
        jmp short @@End

@@CommandError:
        call    Beep
        jmp     @@GetAddressOrCommand
@@End:  popad
        ret
ENDP GetAddressOrCommand
ENDS

; Подключить подпрограмму, переводящую сегментный
; регистр GS в режим линейной адресации
include "lst_3_01.inc"
; Подключить набор процедур вывода/вывода данных
include "lst_2_02.inc"

END
```

Метод Родена проверен не только на процессорах Intel, но и на клонах, изготовленных AMD, Cyrix, IBM, TI [9]. На всех протестированных компьютерах переход в режим линейной адресации данных проходил нормально, то есть метод не только работоспособен, но и универсален. Метод Родена в свое время не был оценен по достоинству, поскольку обычный объем памяти персональных компьютеров составлял тогда 1-2 Мбайт, и преимущества линейной адресации не были очевидными. Резкое увеличение объема памяти в устройствах массового применения произошло гораздо позже — начиная с 1995 года. В это же время был внедрен новый стандарт на видеоконтроллеры (VESA 2.0) и появилась возможность линейной адресации видеопамяти, однако о методе Родена программисты уже успели забыть. Между тем, совместное использование линейной адресации данных в оперативной памяти и линейного пространства видеопамяти дает наибольший выигрыш по скорости выполнения программ и позволяет сильно упростить алгоритмы построения изображений.

Таким образом, метод Томаса Родена обладает следующими основными преимуществами [9]:

- имеется свободный доступ ко всем аппаратным ресурсам компьютера;
- возможна линейная адресация всей оперативной памяти и памяти видеоконтроллера;

- логические и физические адреса отображенной на шину процессора памяти периферийных устройств совпадают;
- метод совместим с клонами процессоров Intel;
- сохраняется возможность использования всех функций DOS и BIOS, как в обычном реальном режиме работы процессора.

Последнее свойство особенно важно: не нужно разрабатывать собственные программы для работы с периферийными устройствами на уровне регистров, следовательно, не проявляются и не создают лишних проблем нестандартные особенности оборудования.

Основной недостаток метода Родена — существенное ослабление защиты памяти. Поскольку отменен контроль границы сегмента данных, работающая с линейным пространством подпрограмма в случае ошибки адресации или зацикливания может не только разрушить смежные данные, но и вообще стереть все содержимое оперативной памяти, в том числе все программы и резидентную часть операционной системы. Чаще всего стирается таблица векторов прерываний, размещенная в начале адресного пространства. Следовательно, необходимо ограничивать число подпрограмм, работающих с линейной адресацией, и очень тщательно их отлаживать.

Второй недостаток прямо вытекает из первого — работа в реальном режиме DOS и ослабление защиты не позволяют реализовать многозадачность. Однако для решения прикладных задач часто вполне достаточно фоново-оперативного режима работы, когда всеми ресурсами системы распоряжается один программный модуль, а остальные предназначены для узкоспециальных целей и вызываются на короткие промежутки времени через механизм прерываний. Иными словами, доступ к видеопамяти и всей оперативной памяти должен быть только у основной программы, а вспомогательные процедуры и драйверы периферийных устройств могут хранить свои данные только в основной области памяти DOS (то есть, в пределах первого мегабайта адресного пространства). Линейная адресация, сама по себе, не накладывает слишком жестких ограничений на работу системы, поскольку персональные компьютеры вообще функционируют в основном в однозадачном режиме: аппаратные средства для реализации многозадачности имеются уже давно, но сильные ограничения создают физиологические и психологические особенности человека, который сидит за компьютером. Любая серьезная работа требует от оператора полной концентрации внимания на одном процессе. То же самое относится к компьютерным играм — невозможно одновременно играть в Quake и редактировать текст.

Третий недостаток: строковые команды процессора x86 в реальном режиме не пригодны для работы с сегментом, настроенным на линейной

адресацию памяти. Это не очень существенный недостаток, поскольку внутренняя RISC-архитектура современных процессоров позволяет выполнять группу из нескольких простых команд с той же скоростью, что и одну сложную составную команду, выполняющую аналогичную операцию. Кроме того, процессор выполняет внутренние операции быстрее, чем операции обращения к оперативной памяти, и гораздо быстрее, чем операции чтения/записи в видеопамять.

В целом можно сказать, что предложенный Роденом режим — это в первую очередь режим учебно-отладочный. Его очень удобно применять в процессе освоения методов непосредственной работы с периферийными устройствами. Во-первых, линейная адресация абсолютно прозрачна — область памяти устройства можно просматривать прямо по физическому адресу. Во-вторых, исследуемое устройство можно рассматривать изолированно, исключив опасность возникновения паразитных взаимодействий с другими аппаратными компонентами и посторонним программным обеспечением.

Далее приведены файлы, включаемые в программу, приведенную ниже [9].

Мнемонические обозначения кодов управляющих клавиш

```
; Для клавиш, традиционно выполняющих определенные
; функции, приведены краткие комментарии справа.

; Для "текстовых" управляющих клавиш вместо скан-кодов
; используются ASCII-коды:
B_RUBOUT  equ    8 ;забой
B_TAB     equ    9 ;табуляция
B_LF      equ   10 ;перевод строки
B_ENTER   equ   13 ;возврат каретки
B_ESC     equ   27 ;"Esc"

; Скан-коды функциональных клавиш:
F1        equ   59 ;вызов подсказки на экран
F2        equ   60
F3        equ   61
F4        equ   62
F5        equ   63
F6        equ   64
F7        equ   65
F8        equ   66
F9        equ   67
F10       equ   68 ;выход из программы

; Скан-коды клавиш дополнительной клавиатуры:
```

```
B_HOME     equ   71 ;перейти в начало
B_UP       equ   72 ;стрелка вверх
B_PGUP     equ   73 ;на страницу вверх
B_BS       equ   75 ;стрелка влево
B_FWD      equ   77 ;стрелка вправо
B_END      equ   79 ;перейти в конец
B_DN       equ   80 ;стрелка вниз
B_PGDN     equ   81 ;на страницу вниз
B_INS      equ   82 ;переключить режим (вставка/замещение)
B_DEL      equ   83 ;удалить символ над курсором

; Скан-коды часто используемых комбинаций клавиш:
ALT_F1     equ  104
ALT_F2     equ  105
CTRL_C     equ    3
CTRL_BS    equ  115
CTRL_FWD   equ  116
CTRL_END   equ  117
CTRL_PGDN  equ  118
CTRL_HOME  equ  119
CTRL_PGUP  equ  122
```

Мнемонические обозначения цветов для цветного текстового видеорежима

```
; "Темные" цвета (можно использовать для фона и текста)
BLACK         equ  0 ;черный
BLUE          equ  1 ;темно-синий
GREEN         equ  2 ;темно-зеленый
CYAN          equ  3 ;бирюзовый (циан)
RED           equ  4 ;темно-красный
MAGENTA       equ  5 ;темно-фиолетовый
BROWN         equ  6 ;коричневый
LIGHTGREY     equ  7 ;серый
; "Светлые" цвета (только для текста)
DARKGREY      equ  8 ;темно-серый
LIGHTBLUE     equ  9 ;синий
LIGHTGREEN    equ 10 ;зеленый
LIGHTCYAN     equ 11 ;голубой
LIGHTRED      equ 12 ;красный
LIGHTMAGENTA  equ 13 ;фиолетовый
YELLOW        equ 14 ;желтый
WHITE         equ 15 ;белый
```

4.2 Вопросы для самопроверки

1. В каком режиме работает микропроцессор x86 сразу после сброса?
2. Как осуществляется сегментная адресация памяти в защищенном режиме?

3. Что такое привилегированные команды процессора?
4. Какие действия надо предпринять, чтобы в программе выполнялись привилегированные команды?
5. Какова структура дескриптора сегмента?
6. Для чего используются сегментные регистры в защищенном режиме?
7. Что такое селектор дескриптора?
8. Что такое линейный адрес?
9. Что такое базовый адрес сегмента?
10. Что такое лимит сегмента?
11. Для чего нужны атрибуты сегмента?
12. Как выполняется перевод процессора в защищенный режим из реального?
13. Что такое псевдодескриптор?
14. Какая команда используется для загрузки таблицы глобальных дескрипторов?
15. Что такое теневые регистры дескрипторов?
16. Почему перед переводом процессора в защищенный режим надо запретить все прерывания?
17. Почему нельзя корректно завершить программу, находясь в защищенном режиме?
18. Когда загружаются теневые регистры дескрипторов процессора?
19. Каким образом обнуляется стек предвыбранных команд при переходе в защищенный режим?
20. Зачем обнуляется стек предвыбранных команд при переходе в защищенный режим?
21. Как производится возврат из защищенного режима работы процессора в реальный?
22. Что такое линейный режим адресации памяти в реальном режиме?
23. С каким объемом памяти позволяет работать режим линейной адресации в реальном режиме?
24. Чем различается организация памяти в реальном, защищенном и линейном режимах адресации?
25. Почему при работе в линейном режиме адресации необходимо включить адресный сигнал *A*20?
26. Можно ли в режиме линейной адресации работать с памятью видеоконтроллера?
27. Можно ли в режиме линейной адресации выполнять функции DOS?
28. Работают ли механизмы защиты памяти при использовании метода линейной адресации в реальном режиме?

Список литературы

1. Гук М. Процессоры Intel от 8086 до Pentium II.– СПб.: 2008. – 224 с.: ил.
2. Пустоваров В.И. Ассемблер: программирование и анализ корректности машинных программ. – К.: Издательская группа BHV, 2008.– 480 с.
3. Брамм П., Брамм Д. Микропроцессор 80386 и его программирование: Пер. с англ. – М.: Мир, 2008.
4. Браун Р., Кайл Дж. Справочник по прерываниям для IBM PC: в 2-х т. Пер. с англ. – М.: Мир, 2014. – Т. 1. – 558 с.; Т.2. – 480 с.
5. Дао Л. Программирование микропроцессора 8088: Пер. с англ. – М.: Мир, 2008. – 357 с.
6. Нортон П. Персональный компьютер фирмы IBM и операционная система MS DOS: Пер. с англ. – М.: Радио и связь, 2011. – 416 с.
7. Юров В. Assembler. – СПб.: Издательство Питер, 2014. – 624 с.: ил.
8. Юров В., Хорошенко С. Assembler: учебный курс. – СПб.: Издательство Питер, 2009. – 672 с.: ил.
9. [Электронный ресурс] – Режим обращения: https://prog-cpp.ru/asm-macro/ (Дата последнего обращения 2 февраля 2017 года)
10. Кулаков В. Программирование на аппаратном уровне. Специальный справочник. – СПб: Питер, 2010. – 496 с.: ил.
11. Thomas Roden. Four Gigabytes in Real Mode. – Programmer’s Journal 7.6, 2009.
12. Intel Architecture Software Developer’s Manual, Volume1: Basic Architecture. – Intel Corp., 2009.
13. Intel Architecture Software Developer’s Manual, Volume1: Instruction Set Reference Architecture. – Intel Corp., 2009.
14. Intel Architecture Software Developer’s Manual, Volume1: System Programming. – Intel Corp., 2009.
15. Гук М. Процессоры Pentium II, Pentium Pro и просто Pentium. – СПб: Питер Ком, 2009. – 288 с.: ил.
16. Рудаков П.И., Финогенов К.Г. Программируем на языке ассемблера IBM PC. Изд. 3-е. – Обнинск: Изд-во «Принтер», 2009, – 495 с.: ил.

Сведения об авторе

Рощин Алексей Васильевич, кандидат технических наук, профессор, профессор кафедры аппаратного, программного и математического обеспечения вычислительных систем Физико-технологического института Московского технологического университета (МИРЭА).