

Материалы к занятию 22 апреля 2020

Тема: работа с большой памятью в реальном режиме.

Это, пожалуй, самая сложная и самая интересная тема. Основываясь на материалах (и программах) этой темы можно в прикладных программах реального режима использовать для хранения данных всю имеющуюся оперативную память, вплоть до 4 Гб (разумеется, если рабочий компьютер выполнен на процессоре 386+). Думаю, вряд ли кто из современных студентов живьем видел компьютер с процессором более младшего поколения.

Суть дела смотрите в пособии, на странице 217.

Коротко суть заключается в том, что в момент создания **DOS** существовал лишь базовый процессор 8086/88, который имел четыре сегментных регистра и работал в режиме, который позднее стали называть «реальным». Процессор 80186 существенного влияние на РС не оказал, так как оказался не полностью совместимым с базовым процессором. Процессор 80286 позволил работать с оперативной памятью 16 мегабайт (разумеется в другом режиме, который назывался защищенным, а предыдущий режим стал называться реальным). Защищенным режим назвали потому, что, если в DOS (в однозадачной операционной системе) пользователь получал монопольный (и полный) доступ ко всем ресурсам (в первую очередь, к аппаратуре), то в многозадачной и/или многопользовательской системе это недопустимо. Именно поэтому в процессор ввели всевозможные механизмы защиты операционной системы от пользователей, пользователей друг от друга, а также всего этого «от дурака». Так в процессоре было введено понятие «привилегия», то есть, «уровень допуска», который определял возможности работы субъекта с данной привилегией с различными ресурсами. Хотя в процессоре было определено четыре уровня привилегий (от 0 до 3, где 0 – наивысший уровень «допуска»), в последствии в операционных системах использовались только 2 уровня, что связано с желанием распространить разрабатываемые операционные системы (в частности, Windows) платформы, среди которых встречались варианты всего с двумя уровнями.

В итоге (в частности, в Windows) различают уровень ядра (0 уровень) и пользовательский уровень (3 уровень). Первый уровень защиты определяется дескриптором сегмента, в котором указан его уровень привилегий (см. пособие OVB2 на сайте). Если к сегменту обращается некоторый код, то процессор сравнивает уровень привилегий запроса и уровень привилегий сегмента и, при недостаточном уровне запроса, он отвергается. Кроме того, имеются «технические» методы защиты. Например, в сегмент кода никогда нельзя ничего записать, а читать из него можно только в том случае, когда у него взведен атрибут «разрешение чтения». Сегмент данных никогда не может быть исполнен, а запись в него возможна лишь в том случае, когда у него взведен атрибут «разрешение записи». И далее см. пособие OVB2.

Процессор 80286 прожил недолгую активную жизнь, на смену ему пришел процессор 80386, который (как выяснилось впоследствии) оказался настолько хорош, что процессор Pentium, разработанный существенно

позднее, был реализован фактически, как два 386 процессора, «сидящих» на 64-разрядной шине. Все последующие процессоры линейки x86 повторяют эту логическую структуру. В этом процессоре были реализованы три режима: «реальный», защищенный и режим виртуальной реальной машины 8086. Здесь реальный заключен в кавычки не просто так. Этот реальный режим на самом деле является лишь эмуляцией реального режима (с 1 Мб памяти), на чем и основан материал этого занятия.

Тот факт, что реальный режим в 386 процессоре является не настоящим, а эмулируемым, позволяет использовать описанный ниже механизм. Механизм основан на том, что в этом процессоре для работы используются теневые сегментные регистры, в которых записаны база (начальный адрес сегмента) и лимит (его размер – см. пособие OVB2). Они программно недоступны, а запись в них производится в тот момент, когда в программно доступный сегментный регистр осуществляется запись. Таким сложности пришлось сделать из-за того, что иначе нельзя было бы переходить из реального режима в защищенный и наоборот (из-за различных методов использования сегментных регистров – в реальном режиме для хранения сегментного адреса, а в защищенном – для хранения селектора, указывающего на запись в таблице дескрипторов).

При эмуляции реального режима в этом процессоре в теневых сегментных регистрах хранятся база и лимит, соответствующие реальному режиму (соответственно, 0 и 1 Мб). При переходе в защищенный режим в теневые регистры записываются данные о сегментах, соответствующие защищенному режиму (об этом должен позаботиться программист), а при переходе обратно в реальный режим в теневые регистры надо записать данные, соответствующие реальному режиму. В принципе, мы могли бы оставить в сегментных регистрах **CS**, **DS**, **SS** или **ES** данные, соответствующие защищенному режиму, но это сохранится лишь до запуска какой-либо программы, так как DOS при запуске инициализирует сегментные регистры по-своему (а другого режима, кроме реального, она не знает). Весь фокус заключается в том, что в 386 процессоре кроме, этих четырех сегментных регистров, появились еще два – **FS** и **GS**, о которых **DOS** не подозревает (как не подозревает и о существовании 386 процессора). Это позволяет оставить в теневых регистрах **FS** и/или **GS** данные, соответствующие защищенному режиму (в частности, лимит в 4 Гб), а **DOS** не будет их изменять (см. выше).

- В этом занятии мы этим и займемся. Рассмотрим три программы:
- программу, которая в защищенном режиме устанавливает лимит сегментного регистра **GS** на 4 Гб,
- программу, которая в реальном режиме запишет что-нибудь в дальнюю память (в 127 мегабайт),
- программу, которая считывает из дальней памяти то, что мы записали.

Все программы имеются в папке **04\_22N**, а также в материалах на сайте (там они находятся в папке **20\_03\_25**). Программы в каталоге **04\_22N**

немного отличаются от программ в **20\_03\_25** (там убраны некоторые лишние команды).

Самая сложная (по количеству нового материала) – первая программа, с которой мы и начнем.

Программа называется **lim\_g.asm** (листинг ниже).

Начинается программа со строки 4 (**.386P**). В этой записи буква **P** означает, что можно использовать привилегированные команды (то есть, команды, которые могут навредить системе).

Далее, в строке 5 (**cmos\_a equ 70h**) определяется адрес регистра **A** энергонезависимой памяти, который нам понадобится для запрещения и разрешения немаскируемых прерываний. Название «немаскируемое» означает лишь, что это прерывание нельзя запретить, сбросив флаг **IF**.

Далее, в строках 6-13 определяется структура дескриптора сегмента (см. пособие OVB2).

6 строка (**descry struc**) начинает структуру с названием **descr**. Строка 7 (**lim dw ?**) – первое поле формата «слово», описывающее младшие 16 разрядов лимита (размера сегмента). Напомню, что на лимит сегмента в дескрипторе отведено всего 20 разрядов, которые позволяют непосредственно задать 1 мегабайт, а в страницах – 4 гигабайта. Строка 8 (**base\_1 dw ?**) – младшие 16 разрядов базы (начального адреса) сегмента (всего должно быть 32 разряда). 9 строка (**base\_m db ?**) – средние 8 разрядов базы. Строка 10 (**attr\_1 db ?**) – первый байт атрибутов. 11 строка (**attr\_2 db ?**) – второй байт атрибутов (в котором младшие 4 разряда представляют старшие разряды лимита – до 20 разрядов – а старшие 4 разряда – дополнительные атрибуты (см. страницы 14 OVB2). 12 строка (**base\_h db ?**) описывают старшие 8 разрядов базы. Строка 13 (**descr ends**) – окончание описания структуры **descr**. Назначение этой структуры то же, что и в заголовках запросов драйверов – облегчение программирования.

Строка 14 (**code segment use16**) – начало сегмента, но здесь появилось обозначение, которого раньше не было (**use16**). Это указание компилятору, что по умолчанию будет использоваться 16-разрядная адресация. Это важно, так как указание **.386** позволяло компилятору решить, что адресация будет 32-разрядная. При этом можно использовать 32-разрядную адресацию, но это надо специально указывать (что мы и будем делать).

Строки 15 и 16 (**assume cs:code,ds:code, Org 100h**) вопросов вызвать не должны.

В строке 18 (**begin: mov rcs,cs**) начинается собственно программа. Обратим внимание на то, что точкой входа в программу будет метка **begin**. Эта команда готовит возврат в реальный режим из защищенного. Как это делается будет видно далее.

В строке 19 (**call mes1**) вызывается процедура печати начального сообщения программы, а в строке 20 (**jmp work**) выполняется обычный перескок через переменные.

Далее идут переменные. Первым в переменных стоит псевдодескриптор. Прежде, чем говорить о нем, скажем, для чего он нужен.

Как известно, в защищенном режиме все сегменты, которые предполагается использовать в сеансе операционной системы, должны быть описаны в глобальной таблице дескрипторов **GDT**. Эта таблица должна находиться в оперативной памяти (в заблокированной области, то есть, она не может выгружаться на диск). Параметры **GDT** (ее длина и базовый адрес) должны быть загружены в специальный регистр процессора (регистр глобальной таблицы дескрипторов). Для загрузки этого регистра используется специальная команда **lgdt (Load Global Descriptor Table)**. Синтаксис команды: **lgdt источник**. В качестве операнда в команде указывается адрес области в формате 16+32. Младшее слово области — размер **GDT**, двойное слово по старшему адресу — значение базового адреса начала этой таблицы. Эти компоненты формируются в памяти заранее. Переменная **pdescr**, которой начинается область переменных (строка 18) и является тем источником, откуда будет загружаться регистр **GDT**.

Итак, строка 22 (**pdescr dw ?**) — длина глобальной таблицы дескрипторов, строка 23 (**dd ?**) — ее базовый адрес. То есть, переменная **pdescr** состоит из двух полей — двухбайтового и четырехбайтового. Можно было бы как и для дескриптора определить структуру, но в силу простоты переменной, не станем этого делать.

Далее, со строки 25 формируем глобальную таблицу дескрипторов **GDT**. Глобальная таблица дескрипторов должна начинаться с нулевого дескриптора (со всеми нулевыми полями), к которому обращение запрещено (это еще одна из степеней защиты). Так как выше мы определили структуру дескриптора, то воспользуемся записью структуры в угловых скобках, о которой мы говорили на прошлом занятии.

Итак, строка 25 (**gdt\_null descr <0,0,0,0,0,0>**) определяет нулевой дескриптор. Смысл этой записи такой:

- **gdt\_null** — название формируемой переменной,
- **descr <0,0,0,0,0,0>** — присвоение значений полям этой переменной.

В угловых скобках через запятую перечисляются поля структуры. Здесь надо помнить, что хотя все нули в этой записи выглядят одинаково, но первые два нуля шестнадцатиразрядные, а остальные — восьмиразрядные (согласно определению структуры выше). Если инициализировать какие-то поля не надо, то эти поля можно пропустить (запятые при этом должны присутствовать). Например, если надо инициализировать только второе поле, остальные оставить неизменными, можно записать так: **descr <,0,,,>**. Если ни одно поле инициализировать не надо, то можно просто записать **descr <>**. При этом будет создана переменная и отведена необходимая память.

Значение селектора (значение в сегментном регистре), по которому мог бы быть выбран этот сегмент — 0. Напомним еще раз, что обращение к нулевому сегменту запрещено.

Строка 26 (**gdt\_code descr <0ffffh,0,0,9ah,0,0>**) — дескриптор сегмента кода. Разберем значения полей подробно.

Поля первое (**0ffffh**), и младшие четыре бита пятого поля (**0**) характеризуют лимит (размер сегмента. Мы видим, что старшие четыре бита

нулевые, а младшие – все единицы, то есть, значение лимита =64К. Осталось выяснить, в каких единицах лимит задан, в байтах, или в страницах. Это зависит от атрибута гранулярности **G**, который описывается старшим битом этого же пятого поля. Так как весь байт этого поля нулевой, значит атрибут гранулярности сброшен, и размер сегмента задается в байтах. В итоге мы получаем размер сегмента 64 килобайта, что и соответствует реальному режиму. Теперь смотрим, как задан базовый адрес (мы пока не знаем этого адреса, поэтому пока заполняем нулями). Это поля: второе (**0**), третье (**0**) и шестое (**0**). Так как все они нули, значит базовый адрес – нулевой.

Осталось невыясненным четвертое поле – **attr1** (байт атрибутов). Значение у него **9ah** или, в двоичном виде 10011000. Смотрим в пособие (OVB2, страница 14). Тип сегмента определяется тремя битами 3, 2, 1, то есть у нас 100, что означает сегмент кода без возможности чтения. В старшей тетраде взведены биты седьмой (присутствие в памяти) и четвертый (говорящий о том, что данный дескриптор описывает именно сегмент в памяти).

Для обращения к этому сегменту в сегментном регистре должен быть селектор 8. Для чего нам нужен дескриптор сегмента кода. Пока мы работаем в защищенном режиме, все происходит естественным путем. Однако, после перехода в защищенный режим, чтобы выполнять ту же программу, мы должны иметь возможность адресовать ту же область памяти, но уже методами защищенного режима. Для этого мы и описываем область расположения нашей программы дескриптором защищенного режима. Напомню, что селектор этого сегмента – 8.

Строка 28 (**gdt\_gs descr <0ffffh,0,0,92h,8fh,0>**) определяет дескриптор для сегментного регистра **GS**, то есть, того сегментного регистра, с помощью которого мы будем «лазить» в дальнюю память.

Разберем поля этой структуры. Поля первое (**0ffffh**), и младшие четыре бита пятого поля (**0fh**) характеризуют лимит (размер сегмента). В данном случае, это все (20) единицы, а старший бит этого поля (бит гранулярности **G**) взведен, следовательно, размер задан в страницах по 4 килобайта. В итоге получаем размер сегмента, равный 4 гигабайтам. Базовый адрес, как и в случае сегмента кода, нулевой. Байт атрибутов (четвертое поле) **92h**, то есть, тип сегмента 001, что означает сегмент данных с возможностью чтения и записи. В старшей тетраде взведены биты седьмой (присутствие в памяти) и четвертый (говорящий о том, что данный дескриптор описывает именно сегмент в памяти). Селектор этого сегмента – 16 (или **10h**).

Дескрипторы для сегмента данных и для сегмента стека мы формировать не будем, так как ничего, связанного с этими сегментами в защищенном режиме делать не собираемся. В итоге вся глобальная таблица дескрипторов состоит из трех записей.

В строке 29 (**gdt\_size=\$-gdt\_null**) уже известным способом (по резидентным программам) определяется размер глобальной таблицы дескрипторов.

Далее, в строке 30 (**work: mov ax,cs**) находится метка **work**, на которую ведет прыжок через переменные.

Здесь мы должны вычислить линейный адрес начала нашей программы (в **CS** находится ее сегментный адрес). Для вычисления линейного адреса пересылаем сегментный адрес в **AX**. Так как нам нужен будет 32-разрядный линейный адрес, в строке 31 (**movzx eax,ax**) расширяем 16-разрядное значение до 32-разрядного, заполняя остальное пространство нулями. Команда **movzx** позволяет переслать значение из операнда меньшего размера в операнд большего размера, заполняя все «лишнее» пространство нулями. Исходный операнд может быть байтом или словом, а операнд назначения – словом или 32-разрядным. В данном случае мы просто расширяем **AX** до **EAX**.

Так как у нас в **AX** сегментный адрес, то для получения линейного адреса его надо умножить на 16 (размер параграфа), что мы и делаем в строке 32 (**shl eax,4**), сдвигая содержимое **EAX** на 4 двоичных разряда влево.

В строке 33 (**mov ebp,eax**) сохраняем получившееся значение в регистре **ebp** (нам еще пригодится).

В строке 34 (**mov gdt\_code.base\_1,ax**) записываем младшие 16 разрядов линейного адреса из **AX** в соответствующее поле дескриптора сегмента кода. Еще у нас могут быть ненулевыми только 4 разряда (так как мы сдвигали 16-разрядное значение влево именно на 4 разряда). Чтобы получить доступ к старшему слову **EAX** (программного доступа к нему нет), мы в строке 35 (**ror eax,16**) меняем местами содержимое старшего слова **EAX** и **AX**, циклически сдвигая **EAX** на 16 разрядов (направление сдвига не имеет значения).

В строке 36 (**mov gdt\_code.base\_m,al**) мы записываем средние 8 разрядов базы в соответствующее поле дескриптора сегмента кода. Как мы уже говорили, в старших 8 разрядах могут быть только нули, это поле мы не заполняем (там и так нули).

Теперь надо определить линейный адрес начала глобальной таблицы дескрипторов. Для этого в строке 38 (**mov ax,offset gdt\_null**) мы записываем смещение начала этой таблицы в **AX**, затем в строке 39 (**movzx eax,ax**) уже известным способом расширяем это значение до 32-разрядного, а потом, в строке 40 (**add ebp,eax**), добавляем его к сохраненному ранее линейному адресу начала программы, получая в регистре **ebp** линейный адрес начала глобальной таблицы дескрипторов.

Теперь можно заполнить псевдодескриптор. В строке 41 (**mov word ptr cs:pdescr,gdt\_size**) заполняем поле размера глобальной таблицы дескрипторов (**word ptr** здесь обязательно, так как **gdt\_size** константа неизвестного компилятору размера).

В строке 42 (**mov dword ptr es:pdescr+2,ebp**) мы заполняем поле линейного адреса глобальной таблицы дескрипторов. Может возникнуть вопрос: зачем в этой команде стоит **dword ptr**, хотя регистр **ebp** однозначно указывает на 32-разрядный размер. Все дело в том, что структура псевдодескриптора у нас самодельная, и у метки **pdescr** обозначено **dw**. То, что далее идет 32-разрядное поле, знаем только мы. Поэтому, если не указать

**dword ptr** будет ошибка «несовпадение типов». Этого можно было бы избежать, определи мы ранее структуру псевдодескриптора стандартным образом.

Далее идет подготовка к переходу в защищенный режим.

Как известно, все процедуры обработки прерываний, содержащиеся в системном ПЗУ, написаны для реального режима адресации, поэтому в защищенном режиме они работать не могут. Если предполагается, что в защищенном режиме будут работать прерывания, надо подготовить для каждого прерывания соответствующую процедуру защищенного режима (что и делается, например, в ОС **Windows**). Мы работать с прерываниями в защищенном режиме не собираемся, а находиться в защищенном режиме будем менее одной миллисекунды, поэтому прерывания просто запретим. Надо помнить, что запрещать надо не только маскируемые прерывания (которые запрещаются флагом **IF**), но и немаскируемые, которые ресурсами самого микропроцессора запретить нельзя. Зато можно запретить прохождение сигнала немаскируемого прерывания на процессор. Итак, запрещение прерываний.

В строке 44 (**cli**) запрещаются маскируемые прерывания (которые обычно называются просто прерываниями) простым сбросом флага разрешения прерываний.

Для запрещения немаскируемых прерываний необходимо установить старший бит 8-разрядного регистра энергонезависимой памяти **cmos\_a** (адрес которого мы установили константой **70h** ранее). Это мы и делаем в строках 46-51.

В строке 46 (**in al,cmos\_a**) содержимое регистра **cmos\_a** считывается в **AL** (надо помнить, что кроме интересующего нас бита, там есть другие важные биты, которые трогать нельзя).

В строке 47 (**mov ah,al**) мы сохраняем содержимое регистра **cmos\_a** в **AH** (которое затем пересохраним в регистре **CH**, так как аккумулятор активно используется) для того, чтобы при выходе из программы восстановить то состояние старшего бита, которое было перед запуском программы.

В строке 48 (**or al,080h**) мы принудительно взводим старший бит **AL** при помощи команды **ИЛИ** (не трогая остальных битов), а в строке 49 (**out cmos\_a,al**) записываем модифицированное содержимое **AL** в регистр **cmos\_a**. С этого момента немаскируемые прерывания тоже запрещены.

В строке 50 (**and ah,080h**) мы выделяем в сохраненном содержимом регистра **cmos\_a** старший бит (при помощи команды **И**), а в строке 51 (**mov ch,ah**) сохраняем его в регистре **CH**, который портить не собираемся.

В строке 52 (**lgdt fword ptr pdescr**) сформированный нами ранее псевдодескриптор загружается в регистр глобальной таблицы дескрипторов (команда **lgdt** предназначена специально для этого). Определение **fword ptr** говорит о том, что мы имеем шестибайтовый операнд.

Далее в строке 53 (**mov bx,cs**) мы сохраняем содержимое **CS** реального режима в регистре **BX** (чтобы записать его при возвращении в реальный режим в **DS**).

После этого в строках 54-56 выполняется переход в защищенный режим. В строке 54 (**mov eax,cr0**) мы читаем в **EAX** содержимое 32-разрядного управляющего регистра процессора **CR0** (младший бит которого разрешает защищенный режим). В строке 55 (**or al,01**) командой **ИЛИ** принудительно взводим младший бит, не трогая остальные биты, (хотя у нас содержимое **CR0** хранится в **EAX**, бит мы взводим в **AL**, так как другие старшие биты мы не трогаем). В строке 56 (**mov cr0,eax**) записываем измененное содержимое **EAX** (с взведенным младшим битом) обратно в **CR0**. Начиная с этого момента, процессор находится в защищенном режиме.

Теперь проблема заключается в том, что, хотя мы находимся в защищенном режиме, в теновом регистре **CS** находится сегментный адрес реального режима. Для того, чтобы преодолеть это несоответствие надо записать в теневой регистр параметры дескриптора сегмента кода из глобальной таблицы дескрипторов. Для этого надо произвести запись в сегментный регистр, но **CS** программно недоступен, и единственный способ записать что-либо в **CS** – это выполнить дальний переход. Если мы напишем мнемонику дальнего перехода **JMP** (не забываем, что мы находимся уже в защищенном режиме), то компилятор подставит в качестве селектора сегмента (опять же, защищенный режим) содержимое тенового регистра **CS**, в котором вместо селектора находится сегментный адрес реального режима. Это противоречие решается достаточно хитрым образом. В строках 59-61 формируется команда дальнего перехода (в защищенном режиме она имеет формат <Код операции (**0eah**)><32-битное смещение><16-битный селектор>. Итак: 59 строка (**0EAh**) – код команды дальнего перехода, 60 строка (**dw offset pmode**) – метка следующей команды, 61 строка (**dw 8**) – селектор кода сегмента в глобальной таблице дескрипторов. То есть, формат команды для защищенного режима получился правильный, в теневой регистр **CS** будут загружены параметры дескриптора сегмента кода, и далее выполнение команд будет адресоваться именно в этом сегменте.

Итак, мы пришли на метку **pmode** (строка 62 – **pmode: mov ax,16**), где мы делаем то, что хотели сделать в защищенном режиме (записать в теневой регистр сегментного регистра **GS** лимит в 4 гигабайта). То есть, в строке 62 в регистр **AX** записывается селектор **16** или **10h** (это третья запись в глобальной таблице дескрипторов, которую мы сформировали для регистра **GS**), а в строке 63 (**mov gs,ax**) записываем этот селектор в сегментный регистр **GS**, что автоматически вызывает запись параметров третьей записи глобальной таблицы дескрипторов в теневой регистр **GS**. Таким образом, в защищенном режиме мы сделали все, что хотели.

В строках 64-66 выполняется переход в реальный режим:

- строка 64 (**mov eax,cr0**) – чтение в **EAX** содержимого **CR0**,
- строка 65 (**and al,11111110b**) – командой **И** принудительно сбрасываем младший бит, не трогая остальные биты,



- строка 66 (**mov cr0,eax**) – записываем измененное содержимое **EAX** (со сброшенным младшим битом) обратно в **CR0**.

С этого момента мы находимся в реальном режиме (притом, что в теновом регистре находятся параметры сегмента кода защищенного режима).

Теперь надо вернуться в реальный режим, имея в виду вышесказанное. Это противоречие решается также, как в строках 59-61. А именно:

- строка 68 (**0EAh**) – код команды дальнего перехода (но уже формата реального режима – следующее слово 16-разрядное смещение, далее 16-разрядный сегментный адрес),
- строка 69 (**dw offset rmode**) – метка следующей команды,
- строка 70 (**rds DW ?**) – сегментный адрес сегмента кода реального режима (мы сохранили его сюда в строке 18 в начале программы).

Теперь мы можем выполнять команды реального режима, так как восстановлена его сегментная адресация.

Строка 72 (**rmode: movss,bx**) – метка, куда мы перешли предыдущей командой. Здесь мы записываем в сегментный регистр стека **SS** содержимое **CS** реального режима (мы ранее, в строке 53, сохранили его в **BX**).

Строка 73 (**mov ds,bx**) – то же самое делаем с регистром сегмента данных **DS**.

В строке 74 (**xor ax,ax**) – обнуляем (экономным образом) аккумулятор, и далее в строках 75-77 записываем этот нуль в сегментные регистры **ES**, **FS** и **GS** – команды **mov es,ax**, **mov fs,ax** и **mov gs,ax**, соответственно. Сегментные регистры **ES** и **FS** нам нужны не будут, но мы их обнулили на всякий случай.

Мы все подготовили, осталось разрешить прерывания. Сначала немаскируемые прерывания:

- строка 79 (**in al,cmos\_a**) – читаем содержимое регистра **cmos\_a** в **al**,
- строка 80 (**and al,07Fh**) – с помощью команды **И** принудительно сбрасываем старший бит **AL** (во втором операнде все биты, кроме нулевого старшего – единицы),
- строка 81 (**or al,ch**) – с помощью команды **ИЛИ** устанавливаем ранее сохраненный (в строке 51) бит разрешения немаскируемого прерывания,
- строка 82 (**out cmos\_a,al**) – записываем содержимое **AL** в регистре **cmos\_a**.

В строке 83 (**sti**) взводим флаг разрешения маскируемых прерываний.

В строке 85 (**call mes2**) вызываем процедуру вывода второго сообщения программы (фиксирующую возврат в реальный режим).

Строка 86 (**int 20h**) – выход из программы.

Служебные процедуры вывода сообщений одинаковы. Они используют функцию 9 системного прерывания для вывода строки на экран. Отличаются они лишь строками загрузки смещения сообщения в **DX**. Пояснений эти процедуры, я думаю, не требуют.

Строки 95 и 96 – тексты первого и второго сообщений. Здесь не надо забывать добавлять в конце строки терминатор сообщения (\$).

Строки 97 и 98 – стандартное завершение программы (также без пояснений).

#### Листинг **lim\_g.asm**.

```
; Программа для снятия предела с сегментного регистра
; GS, что позволяет в реальном режиме работать с
; данными за пределами 1 МБ
.386P
cmos_a equ 70h
descr struct ;Структура дескриптора сегмента
lim dw ?; Граница (биты 0 - 15)
base_1 dw ?; База (биты 0 - 15)
base_m db ?; База (биты 16 - 23)
attr_1 db ?; Байт атрибутов 1
attr_2 db ?; Граница (биты 16 - 19) и атр. 2
base_h db ?; База (биты 24 - 31)
descr ends
code segment use16
assume cs:code,ds:code
Org 100h

; Setup Environment
begin: mov rcs,cs ; Для возвр.в реальн.реж.
call mes1;Вывод начального сообщения
jmp work ;Перескок через переменные

;Variables
pdescr dw ? ;Длина GDT Псевдодескриптор
dd ? ;Лин.адр.GDT

;Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0>;Сел.0-обяз.дескр.
gdt_code descr <0ffffh,0,0,98h,0,0>;Селектор 8 -
;P=1-в памяти,S=1-сегмент,тип=100-кода без разр.чтения
gdt_gs descr <0ffffh,0,0,92h,8fh,0>;Селектор 16 для GS
gdt_size=$-gdt_null ; Размер GDT
work: mov ax,cs ; Выч.линейного адреса CS
movzx eax,ax ;Расширение с заполнением нулями
shl eax,4 ;Преобр.из сегм.адр.в линейный
mov ebp,eax ; Сохр.лин.адр.CS
mov gdt_code.base_1,ax ; 0-15 разр.базы
ror eax,16 ;swap words
mov gdt_code.base_m,al ; 16-23 разр.базы
;Линейный адрес и лимит GDT для GDTR
mov ax,offset gdt_null;Смещ.GDT
movzx eax,ax ;Лин.адр.GDT
add ebp,eax ;EBP=Лин.адр.GDT
mov word ptr cs:pdescr,gdt_size ;Заполнение
mov dword ptr es:pdescr+2,ebp ;псевдодескриптора
;Запрещение прерываний в защищенном режиме
cli ;Запрещение маскируемых прерываний
;Запрещение немаскируемых прерываний
in al,cmos_a;Регистр КМОП памяти
mov ah,al ;Чтение регистра
```

```

    or    al,080h    ;Не трогаем остальные биты
    out   cmos_a,al;Запись в регистр 70h
    and   ah,080h    ;Выделение бита разр.NMI
    mov   ch,ah      ;Сохр.бывшего сост.маски NMI
    lgdt  fword ptr pdescr ;Загр.GDTR
    mov   bx,cs      ;BX=CS
    mov   eax,cr0    ;Упр.регистр CR0
    or    al,01      ;Уст.бита разр.защ.реж.PE
    mov   cr0,eax    ;Разреш.защ.режима
;***** Теперь в защищенном режиме
;   Загрузка теневого регистра CS
    db    0EAh ; Код прямого дальнего перехода
    dw    offset pmode ; Смещение
    dw    8     ; Селектор сегмента кода
pmode: mov    ax,16 ;Селектор GS
    mov    gs,ax ;Запись в теневой регистр GS
    mov    eax,cr0 ;Упр.регистр CR0
    and    al,11111110b ; Сброс бита PE
    mov    cr0,eax ; Запрет защищ.режима
;   Эмуляция дальнего перехода на метку rmode (защита от
; компилятора)
    DB    0EAh ; Прямой дальний переход
    DW    (OFFSET rmode) ; По этому смещению
rcs DW    ? ; По этому сегментному адресу
;***** Теперь в реальном режиме
rmode:  mov   ss,bx      ; Возвращение SS=CS
    mov   ds,bx ; DS=CS
    xor   ax,ax ; Обнуление AX
    mov   es,ax ; ES=0
    mov   fs,ax ; FS=0
    mov   gs,ax ; GS=0 (было GS=16)
;   Разрешение прерываний в реальном режиме
    in    al,cmos_a ;Разрешение немаскируемых
    and   al,07Fh    ;прерываний
    or    al,ch      ;Восст.бывшего состояния NMI
    out   cmos_a,al
    sti   ; Разрешение маск.прерываний
    call  mes2
    int   20h
; Служебные процедуры
mes1:    lea   dx,msg1
    mov   ah,9
    int   21h
    ret
mes2:    lea   dx,msg2
    mov   ah,9
    int   21h
    ret
msg1 db    'Запущена программа для снятия лимита с GS',0dh,0ah,'$'
msg2 db    'Лимит с GS снят',0dh,0ah,'$'
code    ends
        end          begin

```

Следующая программа (**f\_write.asm** – дальняя запись) записывает в 127 мегабайт оперативной памяти сообщение, которое мы будем потом читать третьей программой. Эта программа сначала выводит с указанного адреса 255 символов '|', а затем с этого же адреса выводит текстовое сообщение.

Начнем разбирать программу.

Строки с 3 по 6 уже стандартны, и пояснений не требуют. предложение **.386** нужно для того, чтобы, во-первых, можно было использовать сегментный регистр **GS**, а во-вторых, можно было использовать при необходимости 32-разрядную адресацию. Предложение **use16** нужно для того, чтобы по умолчанию адресация была 16-разрядная.

Строка 7 (**sym equ '|'**) определяет символ, которым будет заполняться область в 255 байтов в верхней памяти. В ходе экспериментов с программой можно будет менять этот символ.

Строка 8 (**fadr equ 7000000h**) определяет дальний адрес (это начало 127 мегабайта), с которого начнется запись (в форме константы). Этот адрес также можно будет попробовать менять в ходе экспериментов с программой. При этом надо не забывать менять адрес и в программе, которая читает эти данные.

Строка 10 (**start: mov ebx,fadr**) содержит метку для точки входа и первую команду программы. Этой командой в 32-разрядный регистр **EBX** записывается адрес начала записываемой области.

Строка 11 (**mov cx,255**) инициализирует счетчик числа записываемых символов, а в строке 12 (**mov al,sym**) в **AL** записывается символ заполнения, определенный ранее.

В строке 13 (**st1:mov gs:[ebx],al**) начинается цикл вывода 255 символов в дальнюю область. Здесь метка **st1** для замыкания цикла, а команда **mov gs:[ebx],al** записывает содержимое **AL** в ячейку памяти в сегменте **GS** (префикс переопределения сегмента **gs:**), а 32-разрядное смещение задается в регистре **EBX** (начальное значение **fadr**).

В строке 14 (**inc ebx**) выполняется инкремент (увеличение на 1) адреса ячейки памяти, а в строке 15 (**loop st1**) команда **loop** замыкает цикл (напомню, что команда **LOOP** уменьшает значение в регистре **CX** на единицу, и если после этого значение в **CX** не равно нулю, то команда выполняет переход на указанную метку).

После выхода из цикла управление переходит на строку 16 (**lea bp,mes**), где в регистр **BP** загружается смещение (16-разрядное по умолчанию) метки **mes**, с которой начинаются символы выводимого текстового сообщения.

В строке 17 (**mov ebx,fadr**) опять в 32-разрядный регистр **EBX** записывается адрес начала записываемой области.

В строке 18 (**mov cx,siz**) инициализируется счетчик числа записываемых символов, причем количество выводимых символов вычисляется по реальному сообщению (чтобы можно было его менять, не заботясь об изменении количества выводимых символов).

Со строки 19 (**st2:mov al,[bp]**) начинается цикл записи символов сообщения в памяти (который похож на предыдущий цикл, но в нем выводится не один и тот же символ, а разные, поэтому инкрементируется не только адрес в дальней памяти, но и адрес выбираемого символа). В этой строке опять есть метка для замыкания цикла, а команда записывает в регистр **AL** текущий символ сообщения. Вспомним, что регистр **BP** по умолчанию адресуется в сегмент стека **SS**, но, так как у нас программа типа **.com**, все известные **DOS** сегментные регистры (**CS**, **DS**, **SS** и **ES**) имеют одно и то же содержимое.

В строке 20 (**mov gs:[ebx],al**) содержимое **AL** (текущий символ сообщения) записывается в ячейку дальней памяти в сегменте **GS** (опять префикс переопределения сегмента **gs:**), а 32-разрядное смещение задается в регистре **EBX** (начальное значение опять **fadr**).

В строке 21 (**inc bp**) выполняется переход к следующему символу сообщения, а в строке 22 (**inc ebx**) – переход к следующей ячейке дальней памяти.

В строке 23 (**loop st2**) стандартным образом замыкается цикл записи символов сообщения в дальнюю память.

Строка 24 (**int 20h**) – стандартный выход из программы.

Строки 25 и 26 содержат символы записываемого в дальнюю память сообщения (обратите внимание на то, что в конце нет привычного терминатора – **\$**, так как здесь он не нужен).

В строке 27 (**siz=\$-mes**) определяется длина (количество символов) записываемого в дальнюю память сообщения. Как всегда размер вычисляется как разность между текущим адресом компиляции и меткой начала сообщения.

Последние две строки команды (28 и 29) стандартны, и в пояснениях не нуждаются.

#### Листинг **f\_write.asm**.

```
;Запись в большую память 7000000 (127-ой МБ)
;Здесь баз.адр.в GS равен 0, лимит - 4Гб
.386 ; 11.11.2019
Assume CS: Code, DS: Code
Code SEGMENT use16
    org     100h
sym equ    '|'
fadr equ   7000000h ; (127-ой МБ)
;Запись 255 символов sym в 127-Мбайт
start:  mov ebx,fadr;Лин.удаленный адр.
        mov cx,255  ;Счетчик записи
        mov al,sym  ;Выводимый символ
st1:mov  gs:[ebx],al ;Запись sym в дальний адрес
        inc ebx     ;Следующий байт
        loop st1    ;Цикл заполнения символом sym
        lea bp,mes   ;Адрес выводимого сообщения
        mov ebx,fadr ;Лин.адр.места записи сообщ.
        mov cx,siz   ;Размер сообщения
```

```

st2:    mov al,[bp] ;Символ из сообщ.
        mov gs:[ebx],al ;Далекая запись символа
        inc bp      ;К след.симв.сообщения
        inc ebx      ;К след.симв.на экране
        loop st2     ;Цикл вывода сообщения
        int 20h      ;Выход из программы
mes     db 'Это сообщение записано в память за пределами 1 МБ с '
        db ' адреса 7000000h (Начало 127-го Мбайта) '
siz=$-mes ;Размер сообщения в байтах
code    ends ;Конец сегмента кода
        END         Start ;Точка входа

```

Третья программа комплекта – программа чтения сообщения из дальней памяти **f\_read.asm**.

Эта программа читает из дальней памяти то, что мы записали предыдущей программой.

Начало программы (строки 3-7 ) практически полностью совпадает с началом предыдущей (отсутствует лишь определение символа заполнения, который здесь не нужен).

Первая команда программы (строка 8 – **Start: mov cx,255**) инициализирует счетчик считываемых символов, а также имеет метку для обозначения точки входа в программу.

В строке 9 (**mov ebx,fadr**) в регистр **EBX** помещается 32-разрядное смещение, определяющее начальную ячейку памяти, из которой будут считываться символы.

В 10 строке (**st1:mov al,gs:[ebx]**) начинается цикл считывания символов из дальней памяти. Опять **gs:[ebx]** определяет содержимое ячейки памяти со смещением в **ebx** в сегменте **gs**. Все совпадает с аналогичной командой предыдущей программы, но действие обратное – не запись в память, а чтение из нее.

В строке 11 (**call print**) вызывается процедура вывода на экран символа в режиме телетайпа.

В строке 12 (**inc ebx**) осуществляется переход к следующему байту в дальней памяти, а в строке 13 (**loopst1**) замыкается цикл чтения символов из памяти (напомню, что цикл будет выполнен столько раз, сколько записано в **CX**).

Затем идут ожидание нажатия на клавишу и выход из программы:

- строка 14 (**xor ah,ah**) – нулевая функция клавиатурного прерывания,
- строка 15 (**int 16h**) – клавиатурное прерывание,
- строка 16 (**int 20h**) – выход из программы.

Ниже (строки 17-19) находится процедура вывода символа в режиме телетайпа, которая пояснений уже не требует.

Еще ниже (строки 20, 21) – стандартное завершение программы.

Листинг программы **f\_read**.

```

; Программа вывода на экран содержимого
; памяти с адреса 7000000h (127 Mb)
.386 ; иначе нельзя будет использовать GS
Code SEGMENT use16
    Assume CS: Code, DS: Code
    org 100h
fadr equ 7000000h ; Дальний адрес
Start:  mov cx,255 ; Чтение 255 байтов
        mov ebx,fadr;Лин.дальний адрес
stl:mov al,gs:[ebx];Тек.симв.в AL
        call print ;Печать одного символа
        inc ebx ; К следующему байту
loop stl ; На stl, пока CX не равен 0
xor ah,ah ; Ожидание нажатия любой клавиши
int 16h ;Клавиатурное прерывание
int 20h ;Выход из программы
print:  mov ah,0eh ;Выв.симв.в реж.телетайпа
        int 10h ;Видеоперерывание
        ret ;Возврат
Code    ENDS
        END Start

```

Эксперименты с программами:

- можно попробовать изменить символ заполнения (строка 7 в программе **f\_write**),
- можно попробовать изменить текст сообщения, выводимого этой программой (строки 25, 26),
- можно попробовать изменить адрес, начиная с которого производится запись (строка 8 в программе **f\_write**),
- можно попробовать изменить количество заполняющих символов (строка 11 в программе **f\_write**).

Стандартный порядок запуска программ:

- запуск программы **lim\_g.com**,
- запуск программы **f\_write**,
- запуск программы **f\_read**.

Если попробовать запустить программу **f\_read** или **f\_write** без предварительного запуска программы **lim\_g.com**, эффект будет различным, в зависимости от того, в каком окружении Вы работаете.

Если Вы находитесь в чистой **DOS**, эти действия приведут к перезагрузке операционной системы (из-за попытки выполнить недопустимую команду).

В виртуальной машине эти же действия не вызовут перезагрузки **DOS**, так как в виртуальной машине в сегментном регистре **GS** сняты ограничения на размер сегмента.

**ВАЖНО.** Ни в сеансе **DOS** под **Windows**, ни в оболочке **DosBox** эти программы запустить не удастся, так как это немедленно вызовет ошибку общей защиты (выход по недопустимой команде). Это связано с тем, что ни в

сеансе **DOS** под **Windows**, ни в оболочке **DosBox** не разрешено (строго говоря, запрещено) выполнение привилегированных команд.

Если придумаете еще эксперименты с этими программами, напишите.